TEORÍA PED

PREGUNTAS DE EXAMEN 2023

TEMA 1

1.Diferencia entre Type cast y Type conversion

Cast→ un tipo de datos es convertido en otro por el programador, el tipo de datos destino debe ser más pequeño que el inicial

Conversión → el tipo de datos se cambia automáticamente por otro por el compilador en tiempo de compilación

Diferencia→ en **typecast** el tipo de datos es convertido en otro por el programador mientras que en el **type conversion** es convertido automáticamente en otro tipo de dato en tiempo de compilación por el compilador

TYPE CAST:

```
float num1 = 3.14;
int num2 = (int) num1; // type cast de float a int
```

En este caso, el programador indica explícitamente que se desea convertir la variable "num1" de tipo float a int mediante el uso del operador de type cast (int).

TYPE CONVERSION

```
num1 = 3.14
num2 = int(num1) # type conversion implícita de float a int
```

Consideremos un array de animales:

Ser una subclase→ en todos los sitios que funciona el general tiene que funcionar el específico

No son **covariantes** porque no puedo meter un array de Gatos en un array de perros, entonces Animal [ ] no es covariante a Gato [ ]

No son **contravariantes** porque no podemos asegurar que al leer un elemento de un Animal[] este sea un Gato[], no podemos asegurar que sea del **tipo esperado**

Para ser **bivariantes** tienen que ser covariantes y contravariantes, por tanto la transformación de arrays es invariante o no variante.

Esto se debe a que los arrays son **mutables**, y por tanto, permiten la modificación de sus elementos después de que se hayan creado, por tanto estas reglas no se aseguran

---

T4.PROGRAMACIÓN Y REDES

Atomicity → transacción como una unidad indivisible de operaciones
Consistency→ el estado del sistema después de la transacción funciona
Isolation→ si le llega todo
Durability→ se dan por buenos los cambios

Diferencia entre dirección física y lógica

Dirección física→ depende de la ubicación
Dirección lógica→ es independiente de la ubicación,

Comunicación:

Directa→ interacción explícita de los participantes

Indirecta→ no sabes si el mensaje lo va a coger o no porque utiliza un **medio** para enviar o recibir mensajes, gasta menos direcciones

---

Tubería→ tengo que escribir hasta que se llene, si no escribo y no hay nada escrito espera. Cuando se llena, no puedo escribir más, tengo que esperar a que se vacíe

---

2.Diferencia entre dispositivo carácter y bloque

Los de bloque tienen la **bluff buffer** cache y los de carácter no, para colocarlos de modo bloque necesitamos **caché**.
Caracteres son de forma secuencial 1 a 1 y los bloques en bloques de datos

Bluff buffer cache→ guarda los sectores más utilizados de los discos

Disco, SSD, tarjetas de memoria→ modo bloque
Teclado, ratones, impresoras→ modo carácter

T5.1 FIFOS

¿A qué velocidad va la tubería?

A la velocidad más lenta del que escribe o el que lee

Limitaciones de tuberías→ unidireccional, sólo entre parientes y tamaño fijo del buffer

Se eliminó la de entre padres e hijos creando las **FIFOS** que tengan un nombre y parezcan un fichero, con las mismas llamadas que en los ficheros. Es menos seguro ya que Unix tiene como objetivo la protección

Seguridad→ evitar que te entre alguien y borre

Protección→ evitar que borres algo que es tuyo sin querer, hacerte daño a ti mismo por desconocimiento

Utilizar un usuario normal

PIPE_BUF→ te dice si se puede escribir o no

La petición sea menor que el buffer, para que no se solapen dos peticiones a cachos

Tres llamadas al sistema para abrir la comunicación:

1. Mkfifo
2. Open para leer
3. Open para escribir

¿Por qué las FIFOs son menos seguras que las tuberías clásicas?

Fifos menos seguros que pipes porque depende del **sistema de permisos de UNIX,** medida de protección y no de seguridad.

¿Por qué una FIFO se abre en modo sólo lectura o escritura, pero no en modo lectura/escritura?

Problema del Boca Culo

Lectura y escritura vas a leer lo que acabas de escribir, ya que es una cola y los datos se deben leer en el mismo orden que se escribieron.

Porque es una cola y los primeros datos en entrar son los primeros en salir y para conseguir una **distinción** entre los procesos que proporcionan datos y los que los consumen porque sino causaría **conflictos** entre ambos procesos

¿Qué problemas plantea cada uno de los tres modelos de transferencia de datos en las FIFOS?

1. **Chorro de bytes (byte stream):**

   El principal problema de este modelo es la **falta de delimitadores**. Dado que no hay una estructura explícita de mensajes, el receptor no puede determinar si los datos que recibe fueron escritos **de una vez o en varias operaciones de escritura**. Esto puede dificultar el procesamiento y la interpretación de los datos.

2. **Mensaje con estructura común:**

   Este modelo requiere un **acuerdo previo entre el emisor y el receptor** sobre la estructura del mensaje. Ambos deben conocer y estar de acuerdo en el formato de la estructura común utilizada. Esto implica la necesidad de compartir código o información sobre el formato del mensaje. Si no hay un acuerdo previo o si hay discrepancias en la interpretación del formato, puede haber errores en la comunicación.

3. **Mensaje con delimitadores de fin de mensaje:**

   En este modelo, se utiliza un carácter o secuencia de caracteres como delimitador para indicar el final de cada mensaje. El problema principal radica en el **acuerdo sobre el carácter delimitador.** Si no hay un consenso claro entre el emisor y el receptor sobre qué carácter usar como delimitador, puede haber confusiones y errores en la interpretación de los mensajes.

T5.2 SOCKETS

Modelo cliente-servidor

Misma familia→Mismo tipo de direcciones

UDP y TCP son de la misma familia?

Sí porque utilizan el **mismo tipo de direcciones**

Ambos extremos direccionamiento directo, dirección única de cada socket que dependen del dominio

La ‘ ‘ → Solo funciona como comodín entrante y no saliente

Diferencia entre tubería y socket unix local?

Tuberías son **unidireccionales** solo permite la transferencia en una dirección y los sockets son **bidireccionales,** permiten que los procesos puedan enviar y recibir datos en ambas direcciones

shutdown→ semi cierre, la idea es cerrar la escritura, pero dejar abierta la lectura

En cual no cuesta hacer el close?

UDP no está orientado a conexiones y una vez que los datagramas se han **enviado** se considera que la comunicación ha **finalizado**

Dirección física→ depende de la ubicación de red y mientras que la lógica no

DIFERENCIAS ENTRE UDS Y UDP

En el caso de los sockets UDP, no es necesario verificar si el archivo de socket existe, porque no hay un archivo de socket asociado a ellos como en el caso de los sockets UDS (Unix Domain Sockets). Los sockets UDP se comunican a través de la red utilizando direcciones IP y puertos, por lo que no hay un archivo de socket local en el sistema de archivos.

Tampoco es necesario llamar al método `listen` en el servidor de sockets UDP, ya que no hay conexiones establecidas como en el caso de los sockets TCP. En los sockets UDP, los paquetes de datos se envían y reciben sin necesidad de establecer una conexión antes.

Finalmente, no es necesario cerrar explícitamente el socket del servidor en cada ciclo de bucle en el caso de los sockets UDP. El socket del servidor permanece abierto y escuchando en el puerto especificado hasta que el programa se detiene o el socket se cierra explícitamente. En cambio, en el código del servidor de sockets TCP, el método `accept` bloquea la ejecución del programa hasta que llega una conexión entrante, y es necesario cerrar la conexión establecida después de cada ciclo de bucle.

¿Cuál es el tiempo que domina el de copia o espera?

Conexión de red TCP, el tiempo de **espera** será **indefinidamente** si no escribe. Por lo que habrá que trabajar con esperas y es el tiempo que domina

---

Puedo llamar a una variable con un unicode?

No, sólo se permiten **nº, letras y _** , por tanto no se puede, igual que no se podría llamar a una variable **,** .

Los que figuran como letras, nº o el underscore, el resto no se permite.

PREGUNTAS EXAMEN:

SOCKET_STREAM VS SOCKET_DATAGRAM

Stream→ es más fiable ya que se utiliza **TCP que garantiza las entregas de mensajes enteros,** sin duplicar y sin pérdidas, en cambio el **Datagram**→ envía diferentes datagramas pero pueden llegar desordenados, duplicados o pérdidas

Si tenemos 4 clientes conectados, ¿cuántos sockets tenemos?

Hay 5 sockets, el original + 1 por cada cliente, nº de socket en un servidor activo es = **original + nº clientes conectados**

¿Qué UTF tiene mayor capacidad de representación?

Todas tienen = capacidad, lo que cambia es cómo lo hace cada uno

Ventaja de UTF-8 respecto al resto

Es inmune al problema de **endianness**, mientras que en el 16 y 32 si la otra máquina está en big endian o little se fastidia, como solución existe el BOM para ver el carácter ilegal.

APUNTES CLASE SUELTOS

Diccionario→ obtener el valor a partir de la clave
Si los strings son inmutables, se pueden utilizar como claves ya que no se pueden cambiar

Árbol:No puede haber 2 caminos a un mismo sitio en un árbol, ya que no son direccionales. Propiedad del árbol→cada nodo tiene un padre

Árbol balanceado si solo está vacía la capa final, es decir tienen que estar llenas las capas anteriores

Utilidad→ sirven para las búsquedas

Grafo→ conjunto de nodos conectados por arcos, sin orden.

Utilidad→

En los dinámicos no existe una declaración de datos, es decir, no hace falta poner el tipo de dato. Te lo hace en tiempo de ejecución en vez de compilación como en el estático

UDP

- **Si no se leen todos los datos recibidos el resto se quedan en un búfer esperando su obtención.**
- **Si se recibe más de un paquete la información se concatena en el búfer de recepción, por lo que una llamada a sock.recv() podría leer el contenido de dos paquetes a la vez.**
- **Debido a este último punto, no hay garantía alguna que al usar sock.recv(65495) se obtenga el contenido de únicamente un paquete o incluso, dependiendo de la situación, podría llegarse a leer uno de los paquetes recibidos de manera parcial.**

Select→ permite seleccionar de diferentes fuentes de datos, para no tener colisiones y tener solo un proceso a la vez. Espera a que lleguen los datos

SEÑAL SIGIO
La señal o la ignoras o la gestionas, si la ignoras pierdes los datos que llegan, que es lo que pasa cuando estás haciendo la señal y te llega otro datagrama

E/S asíncrona:

Nos notificará cuando se haya completado la copia, entonces la señal es instantánea

VIERNES TARDE

SABADO TARDE REPASO DE TODO

APUNTES FINAL PED

T1 TECNOLOGÍA DE LA PROGRAMACIÓN

Elección de un lenguaje→ Brooks: a + expresivo - errores (balancear coste del programador / coste de las herramientas)

- Paradigma del lenguaje adecuado al problema y al programador
- + **estructura de datos +** expresividad permite
- Buen **sistema de tipado**

Paradigmas de la programación

Surgen de las ideas de las personas de cómo deben ser construidos los programas

- **Imperativo**→ consiste en acciones para efectuar el **cambio de estado** a raíz de operaciones de asignación o efectos secundarios.
  OOP no es siempre imperativo pero casi todos los OO son imperativos
- **Funcional y Lógica**→ centrado en la **evaluación de funciones**,
  - **Funcionales**→ no todos son puros
  - **Lógica**→ basado en lógica de predicados
- **Concatenativo**→ la construcción de subrutinas se realiza principalmente mediante la **composición de funciones** en lugar de la aplicación de funciones, lo que da lugar a una sintaxis bastante diferente a la de otros lenguajes de programación.

- **Basado en pilas**→ operados por 1 o + pilas cada una con **un propósito**. Las construcciones en otros lenguajes pueden necesitar una modificación para su uso en un programa orientado a pilas
- **Concurrente**→ atraviesa el imperativo, paradigmas orientados a objetos y funcionales
- **Scripting**→ Alto nivel de programación popular en el **desarrollo web**:
  - Desarrollo rápido
  - Tipado dinámico con int,float string y arrays
  - Tipado débil→ una variable x puede ser asignada a un valor de cualquier tipo en cualquier tiempo de ejecución

Conceptos Unificadores:

- **Tipos (**ambos **construidos y definidos por el usuario):**
  - Especificar restricciones en funciones y datos
  - Tipado estático vs dinámico
- **Expresiones**→ aritméticas, booleanas, strings

- **Funciones**→ bloques de código que realizan tareas específicas, se utilizan para dividir el programa en partes más pequeñas y manejables
- **Comandos**→ controlar flujo de ejecución del programa

Entidades de primera clase

1. Creadas en tiempo de ejecución
2. Asignadas a una variable o elemento en una estructura de datos
3. Pasado como argumento a una función
4. Devuelto como resultado de la función

Integers,strings y a veces nº reales, pero clases, funciones y objetos no son siempre entidades de 1ª clase

Estructura de datos

Dato→ Unidades de tratamiento dentro de un sistema informático. Datos **entrada+salida=información**

Componentes de un dato:
- **Identificador**→ nombre para referenciar al dato
- **Tipo**→ rango de valores a tomar
- **Valor**→ elemento que pertenece al rango de valores

Dato lógico→ describe una magnitud con 2 valores/estados, basados en el **álgebra de boole (OR, AND, NOT)**

Endianness→ formato en el que se almacenan los datos de más de un byte.
Little endian: bit + a la izquierda pos + alta
Big endian: bit + a la derecha pos + alta
BI endian: cambia de una a otra con botón HW

Numérico entero (sin signo):

- **Binario puro**

  25 → 00011001
- **BCD**→ agrupaciones de 4 bits por cada decimal

  25 → 0010 0101

Numérico entero (con signo):

- **Binario (Módulo y signo MS):** signo en el bit más a la izquierda

  25 → 0 0011001

  -25 → 1 0011001
- **Complemento a 1 (igual para +, cambio 0s por 1s -):**

```
 25 → 0 0011001
-25 → 1 1100110
```

- **Complemento a 2 (igual para +, negativos C-1 + 1)**

```
 25 → 0 0011001
-25 → 1 1100110 + 1= 1 1100111
```

- **Exceso a 2^n-1:** no hay bit de signo

```
 25 → 128 + 25 = 153  → 10011001
-25 → 128 - 25 = 103  → 01100111
```

Numérico real→ muy grandes o muy pequeños

Representación:

- **Punto decimal—>** 0 a 9 + punto decimal= 25.43
- **Notación científica o exponencial:** nº = mantisa * base exponencial ^característica
  - Mantisa→ nº real
  - Base exp→ base decimal(‘e’)
  - Característica→ exponente correspondiente a un nº entero con su signo

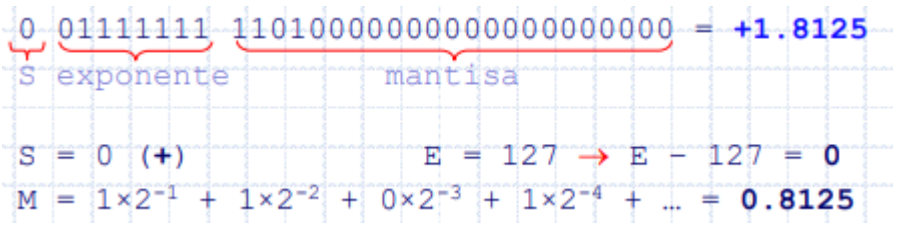
Notación científica o exponencial

Representación interna (IEE 754):

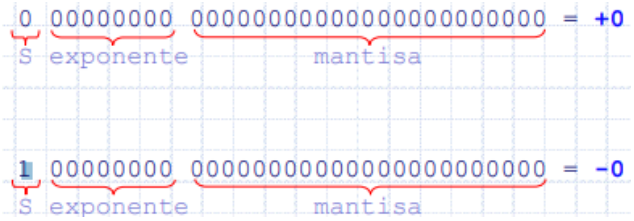
- **Base:** 2 o 10
- **Mantisa:** binario puro
- **Característica:** exceso a 2^n-1 - 1
- **Simple precisión**
  - **Signo**→ 1 bit
  - **Exponente**→ 8 bits
  - **Mantisa**→ 23 bits
- **Doble precisión**
  - **Signo**→ 1 bit
  - **Exponente**→ 11 bits
  - **Mantisa**→ 52 bits

Norma IEE 754

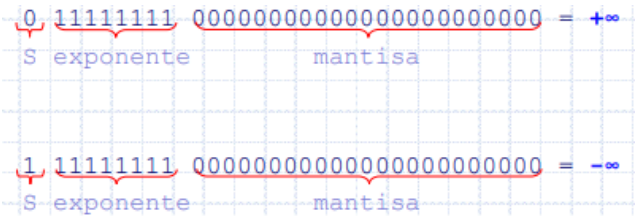
- **Valores Normalizados:** valor V de un número binario de exp E(exceso a n) y mantisa M
  - **Mantisa:** fracción binaria en notación normalizada (sumamos 1 ud asumiendo que el primer bit de la mantisa potencia 0 = 1



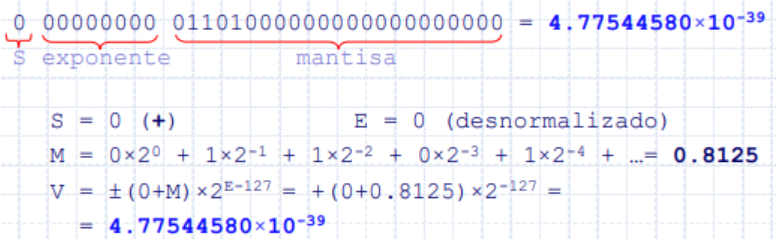
- **Ceros:** no es posible representarlo porque la mantisa va precedida del 1, por convenio:



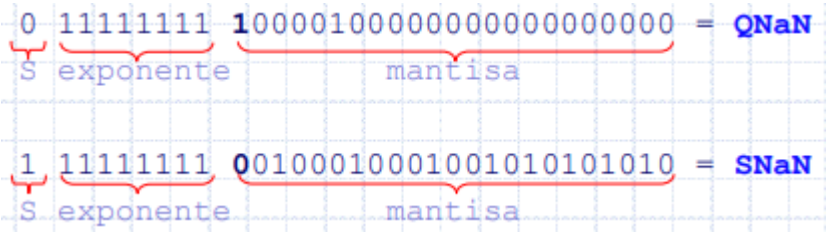
- **Infinitos:** lo mismo que con el 0 por convenio:



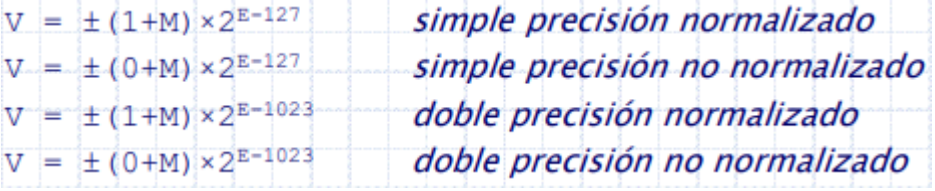
- **Valores Desnormalizados:** no se asume que hay que añadir un 1 a la mantisa:
  - Bits del E =0 M distinta de 0



- **Valores no numéricos: NaN**→ exponente con todo a 1 y M distinta 0
  - **QNaN**→ Primer bit 1 = indeterminado
  - **SNaN**→ Primer bit 0 = Operación no válida



Cálculo de valores:



Datos de carácter:

- Representa un carácter dentro de los definidos por el fabricante: ASCII, EBCDIC, UNICODE
- **ASCII**→ 7 bits almacenados en 8
- **EBCDIC**→ 8 bits

Unicode

Tipos de datos:

- **Derivados**→ para referenciar otro dato con algún medio como la dirección de memoria del otro dato **(puntero)**
    - Tipo→ = al dato que apunta
    - Contenido→ dir del dato al que apunta
    - Indirección→ referencias indirectas al dato que apunta
    - Reutilización→ para apuntar a otro dato del mismo tipo que el puntero
- **Estructurados:**
    - Internos→ mem principal
    - Externos→ mem auxiliar
    - Estáticos→ tamaño no puede cambiar durante su ejecución
    - Dinámicos→ tamaño puede cambiar
    - Lineales→ solo enlazan a un elemento anterior y posterior
    - No lineales→ pueden enlazarse a más de un elemento de diferentes formas
    - Compuestos→ programador mediante agregaciones de datos básicos y derivados
- **Estructuras estáticas:**
    - Tablas→ nº fijo de elementos del mismo tipo en direcciones contiguas: unidimensionales(vectores) , bidimensionales (matrices), multidimensionales (poliedros)
- **Estructuras dinámicas:**
    - **Lista**→ constituida por elementos denominados nodos con **orden significativo** (Contiguas, simplemente o doblemente enlazadas, circulares). Op→ inicializar, insertar eliminar, buscar nodo, copiar o borrar lista, subdividir o concatenar. No acceso aleatorio, insertar o eliminar elemento en posición específico, búsqueda eficiente ya que hay que recorrer todos los elementos
        - **Pila**→ **LIFO:** inicializar,apilar, desapilar, pila llena o vacía y top
        - **Cola**→ **FIFO:** inicializar, encolar, desencolar, cola llena, vacía
    - **Conjunto**→ constituida por elementos **no ordenados**, importa si **pertenece** y si está repetidos los elementos (inicializar, insertar, eliminar, pertenencia, nº de repeticiones de un elemento, conjunto vacío, cardinalidad). No se puede buscar ni ordenar
    - **Diccionario/mapa**→ formada por parejas clave-valor, **orden y repetición** significativo o no (inicializar, insertar, eliminar, lista claves o valor, valor, pertenencia clave) No ordenamiento, acceso aleatorio por posición

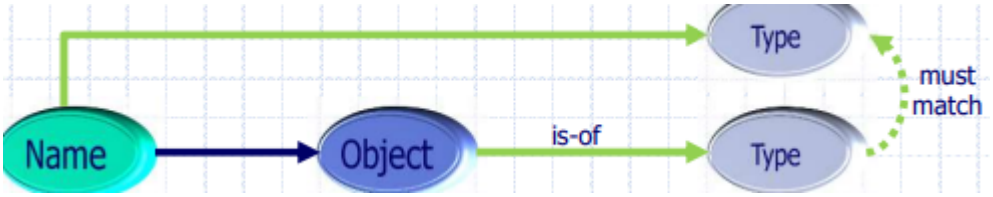
- **Árbol**→ representación de estructuras jerarquizadas. No búsqueda por posición
  - Antecesor, varios sucesores y un único elemento raíz
- **Grafo**→ conjunto de nodos conectados por arcos. No búsqueda por posición, ordenamiento directo
  - Arcos con o sin sentido, cíclicos o acíclicos

Sistemas de tipado

Estipula las formas en las que se deben comportar los programas tipados

Check de tipado estático

- Cada **nombre** está ligado a un **tipo** (a través de una declaración de dato)y al compilar a un objeto
- El vínculo a un objeto es **opcional,** si no está vinculado, name→ **null**
- Cuando una variable está ligada a un tipo, solo puede ser vinculada a objetos de ese tipo o saltará una **excepción**
- Checks en **tiempo de compilación** y check de si el procesamiento del dato es correcto no del dato en sí



Ventajas:

- Refactoring y habilidad de seguir una cadena y saber qué tipo requiere cada paso, + seguros

Check de tipado dinámico

- Cada nombre está ligado a un **objeto** (si no es null) en tiempo de ejecución por medio de asignación de declaraciones
- Es posible vincular un nombre a objetos de **diferentes tipos** en ejecución
- **Errores de tipo** no se detectan hasta ejecutarlo
- Permite más tipos de **construcciones de código**
- Acorta el **editar-compilar-testear-debug**

Ventajas:

- Más conciso y preciso
- Cierres/Funciones
- Leer-evaluar-imprimir loop

Tipado fuerte: no permite que una operación tenga éxito con un tipo distinto

Tipado débil: si lo permite

Lenguaje seguro: no permite operaciones o conversiones que den errores

Lenguaje inseguro: si lo permite

Equivalencia de tipos:

- **Nominativo**→ = nombre
- **Estructural**→ = estructura
- **Duck**→ el lenguaje supone que tipo pretende
-

MIERCOLES MAÑANA

Programación Orientada a Objetos

Clase→ descripción usada para instanciar objetos e identificar propiedades(atributos) que pertenecen a todos los objetos de la clase y comportamientos (métodos)
Objeto→ instancia de la clase con nombre, atributos, valores y métodos
Atributos→ propiedades del objeto
Métodos→ comportamientos
Información escondida y encapsulación→ técnica donde el objeto revela lo mínimo posible de su funcionamiento interno
Asignaciones→ crean o manipulan referencias a objetos: x= y → x referencia al objeto y
Identidad→ pregunta si dos referencia son **exactamente el mismo objeto**
Igualdad→ pregunta si dos referencia son **equivalentes**

Descripción de Kay del OOP: basado en el principio de **diseño recursivo**

- Todo es un **objeto**→ acciones ejecutadas por instancias u objetos
- Objetos realizan computación haciendo **solicitudes** de unos a otros pasándose **mensajes**
    - Las acciones se producen en respuesta a las solicitudes
    - Instancia acepta el mensaje y realiza la acción usando su dato y devolviendo un valor
- **Encapsulación**→ cada objeto tiene su propia memoria que consiste de otros objetos
- Cada objeto es una instancia de una clase y cada clase un repositorio para el comportamiento asociado al objeto

Elementos del OOP:

- **Herencia**→ habilidad para definir clases que son **extensiones** de otras con nuevos atributos y métodos
- **Overriding**→ subclases pueden alterar o sobreescribir info heredada de las clases padre

- **Polimorfismo**→ usar mismo nombre para métodos, permite manipular objetos sin saber su tipo solo su **propiedad común**
- **Overloading**→ operador existente que se le da la capacidad de operar en un nuevo tipo de dato
- **Enlace dinámico**→ proceso de identificar en tiempo de ejecución qué código debe ser ejecutado como **resultado del mensaje**, habilidad de invocar un método de clase derivada vía una clase base

Diseño de OOP (SOLID)

- **S:** principio de **única responsabilidad**→clase solo 1 razón para cambiar
- **O:** principio **abierto-cerrado**→entidades software abiertas a extensión y cerradas a modificación
- **L:** principio de **sustitución liskov**→los subtipos deben ser sustituibles por sus tipos base
- **I:** principio de **segregación de interfaz**→ clientes no deben forzar a depender de métodos que no usan
- **D:** principio de **inversión de dependencia**→ módulos de alto nivel no deben depender de los de bajo nivel, deben depender ambos de las **abstracciones,** al igual que los detalles dependen tb de ellas

Covarianza→ si conserva el orden de los tipos de más específico a menos
Contravarianza→ si el orden es al revés
Bivariante→ si se aplican ambos a la vez A<=B y B<=A
Invariante→ ninguna de las anteriores

Ley de Demeter/No hablar con extraños:

- Cada clase usará un conjunto limitado de otras clases **estrechamente relacionadas**
- Cada clase solo debe enviar mensajes a sus **"amigos"**

T2 SISTEMAS DE CONTROL DE VERSIONES

¿Qué es el control de versiones?

Gestión del desarrollo de cada elemento de un proyecto en el tiempo con posibilidad de añadir, borrar, mover, modificar así como un historial de acciones

Se trabaja sobre un **repositorio** donde se almacena toda la info del desarrollo

Tipos:

Centralizados:

- Repositorio en localización única, necesario acceso a red local o internet

Distribuidos:

- Múltiples copias: copia local para trabajar pero acceso a la red solo para publicar cambios

Centralizados vs Distribuidos
Centralizados: El main repository es el único **fuente** verdadero que contiene todo el historial.
- *Los usuarios checkoutean copias locales de la versión actual*

Distribuidos: Todo checkout del repo es un full repo con su historial completo.
- *Aportan mayor **redundancia y velocidad**. Por tanto se **branchea y merge** más.*

Ciclo de trabajo:

1. Crear copia local-**checkout**
2. Actualizar copia de trabajo -**update**
3. Realizar cambios-**add,delete,copy,move**
4. Examinar cambios-**status,diff,revert**
5. Fusionar cambios-**merge,resolved**
6. Enviar cambios-**commit**
7. Volver al punto 1

Modelos:
1. **Lock-Modify-Lock:** Candado mientras se lee. Leer&Lock, Escribir&Unlock
2. **Copy-Modify-Merge:** Ambos copian, realizan cambios, si hay conflicto merge, si no write

T3 TDD

¿Qué es el TDD?

Es una técnica donde escribes los **casos de test** antes que la implementación del código. Un test no es saber que hacer sino algo que escribir y ejecutar una y otra vez después de aplicarle pequeños cambios

Los módulos deberían tener alta cohesión y bajo acoplamiento
- **Cohesión**→ grado de interacción con el objeto
- **Acoplamiento**→ grado de interacción entre los objetos

¿Por qué usar TDD?

- Es aburrido y requiere a los programadores mantener una exhaustiva serie de test repetibles
- Es más productivo que un debugger
- Boost de confianza cuando sabes hacerlo
- Los desarrolladores pueden trabajar de forma más predecible

Ciclo de vida de un Test

1. **Escribe el test (red)**→ hazlo fallar
2. **Hazlo correr(green)**→ haz que el test funcione rápidamente
3. **Cambia el diseño(refactor)** → simplifica el código

Proceso de TDD

1. Empieza por alguna pequeña funcionalidad
2. Pregúntate ¿qué test demuestran la funcionalidad del código realmente?
3. Hazte una lista con los test a realizar después

RIGHT BICEP→ para validar que los resultados son correctos

- **Boundary (Frontera)**: Meter basura, datos erróneos, nulos, valores con rangos exagerados, duplicados.
- **CORRECT**: Conformance, ordering, range, reference, existence, cardinality y time.
- El tiempo relativo es que cosas suceden antes y que cosas suceden después.
- **Inverse**: si el método realiza algo que tiene operación inversa, aplique la inversa y tener cuidado con los errores que son comunes a ambas operaciones
- **Cross check:** Probar de dos maneras diferentes y que salga el mismo resultado en ambas.
- **Errores**: forzar condiciones de error→ parámetros inválidos, fuera de rango, mock object (objeto de burla)Objetos de mentira que reemplazan a los reales.
- **Performance** (prestaciones): rendimiento absoluto, como cambia el rendimiento a medida que crecen las entradas…

4 técnicas de diseño:

- Cuanto más simple mejor. KISS, YAGNI y OAOO.
- Fake till you make it: primero una constante, después una variable…
- Triangulación: Meter variabilidad suficiente en todos los casos.
- Obvious implementation:

Refactor: Antes del refactor pasan todos los tests y después también. Es una transformación que no cambia la semántica del programa.
Modifica la estructura pero no lo hace

T4 PROGRAMACIÓN Y REDES

Objetivos de primitiva de comunicación:

- Transferencia de datos
- Compartición de info
- Notificación de eventos
- Compartición y sincro de recursos
- Control de procesos

Primitivas entre procesos:

- **Paso de mensajes**→ IPC + básicas
- **Llamadas a procedimientos remotos**→ interacción entre procesos a nivel de lenguaje, comprobación de tipos
- **Transacciones**→ soporte para operaciones entre objetos distribuidos, invocación de métodos remotos

Comunicación entre procesos (IPC)

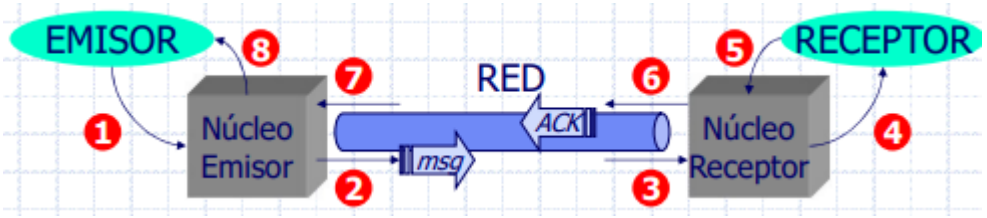
Local o remota, entre dos procesos cualquiera o hijos de = padre

Procesos tienen que comunicarse y sincronizarse

Comunicación entre procesos en Unix

- Ficheros
- Señales→ info es un nº
- Tuberías→ con nombre=FIFOs sin nombre= pipes
- Memoria compartida
    - Espacio del SO
    - Espacio compartido entre procesos (threads)
- Paso de mensajes→ sockets bsd y colas de mensajes

Primitivas de comunicación



- **Envío no bloqueante**→ El emisor continúa al pasar el mensaje al núcleo 1:8
- **Envío bloqueante**→ El emisor espera a que el núcleo transmita por red el mensaje 1:2:7:8
- **Envío bloqueante fiable**→El emisor espera a que el núcleo receptor recoge el mensaje 1:2:3:6:7:8
- **Envío bloqueante explícito**→ igual al anterior pero es la app receptora quien confirma la recepción 1:2:3:4:5:6:7:8
- **Petición-Respuesta**→ El emisor espera a que el receptor procese la op para reanudar la ejecución 1:2:3:4<servicio>:5:6:7:8

Direccionamiento: info válida para la **identificación** de elementos del sistema

- Mecanismos:
    - Dirección **dependiente** de la ubicación (dir física)--> IP+puerto
        - No proporciona transparencia
    - Dirección **independiente** de la ubicación(dir lógica)

- Facilita transparencia
- Necesidad de **proceso de localización** con broadcast o uso de un servidor de localización que mantiene relaciones entre físicas y lógicas
- Uso de **cache** en clientes para evitar localización

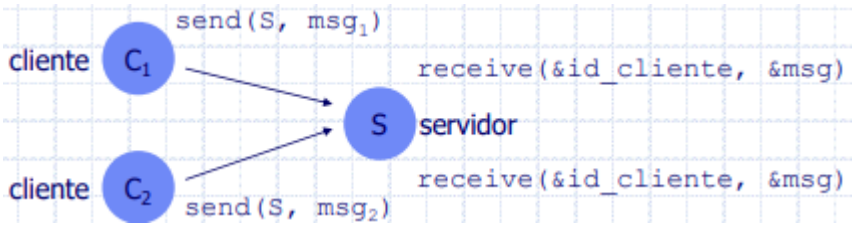
Comunicación directa→ procesos deben nombrar al otro explícitamente

- **Simétrico:** emisor y receptor se conocen explicitamente y establecen conexión

```
send(P, mensaje)
receive(Q, mensaje)
```

- **Asimétrico:** comunicación unidireccional sin conocimiento mutuo explícito

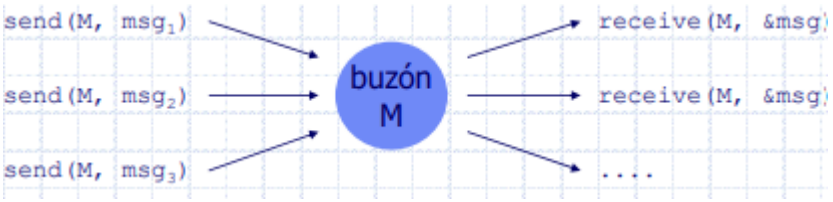
```
send(P, mensaje)
receive(var, mensaje)
```



Comunicación indirecta→ tratar las rutas de comunicación como objetos de primera clase

Buzones:

```
send(M, mensaje)
receive(M, mensaje)
```



Mensajes de texto:

- **Estructura del mensaje**→ cadenas de caracteres `"GET //www.ceu.es HTTP/1.1"`
- **Envío del mensaje** → el emisor debe hacer un análisis de la cadena transmitida

```
send("GET //www.ceu.es HTTP/1.1");
```

Mensajes binarios

- **Estructura del mensaje**

```
struct mensaje_st {
    unsigned int msg_tipo;
    unsigned int msg_seq_id;
    unsigned char msg_data[1024];
};
```

- **Envío del mensaje**

```
struct mensaje_st confirm;
confirm.msg_tipo=MSG_ACK;
confirm.msg_seq_id=129;

send(confirm);
```

Formatos de representación→ para transmisión de binarios emisor y receptor deben coincidir en la interpretación de bits transmitidos

1. Tamaño de los datos numéricos: 16 vs 32 bits
2. Ordenación de bytes: little-endian vs big-endian
3. Formatos de texto: ASCII vs Unicode

Capas de red

- Las **apps** se comunican entre ellas
- La **capa de transporte** tiene que mover bits
- **Capa de red** habla con el siguiente sistema
- **Capa de subred** organiza las tramas para la transmisión, usando los estándares físicos adecuados, las tramas de subred saltan del origen a su destino pasando por los routers

Cada capa usa la que tiene directamente debajo, la inferior añade **cabeceras** a los datos que recibe de la **capa superior.** Parte de los **datos de la capa superior** pueden ser **cabeceras** de capas aún más altas

Modelo TCP/IP→ más sencillo, modelo de 4 capas: aplicación, transporte, internet,enlace de datos

---

JUEVES TARDE

T5.1 TUBERÍAS Y FIFOs

Tuberías→ permiten la transmisión fiable de una cadena o chorro de bytes entre dos procesos (comparable a una cola tipo FIFO)

- Limitaciones→ Half-duplex, usadas entre procesos que tienen un **padre común, tamaño fijo para buffer**
- **Gestión**→ integrada en el **sistema de ficheros**
- Constituyen un **canal de comunicación:**
  - Datos escritos en un extremo se leen en el otro
  - Usa un buffer que define su tamaño máximo
- **Acceso a las tuberías**→ mediante descriptores de e/s de Unix
  - L/E mediante un chorro de bits no estructurado, l y e independientes , permite leer los datos escritos en varias ocasiones

    - Al hacer un **write**→ si hay espacio se escriben los bytes y la llamada retorno, sino la llamada queda bloqueada y la ejecución se suspende hasta que otro proceso haga sitio
- La tubería se crea con pipe(), crea **dos descriptores** uno de escritura y otro de lectura
- **Tamaño del buffer** es **finito,** sólo se escribe hasta una cierta cantidad de bits
- Un proceso creado con **fork**() **hereda** todas las tuberías abiertas del **padre**
- **Procesos y tuberías:**
    - Un proceso crea una tubería y luego hace un fork():
        - Comunicación padre a hijo→ cierra descriptor lectura del padre y escritura del hijo
        - Comunicación hijo a padre→ cierra descriptor de escritura del padre y de lectura del hijo
- **E/S en tuberías:**
    - Buffer lleno→ write(fd_w…) se bloquea
    - Buffer vacío→ read(fd_r…) se bloquea
    - Intenta escribir cuando el **lector cierra**→ SIGPIPE
    - Cuando se **cierra** el extremo **escritor**→ EOF tras recibir los últimos datos
    - Para full-duplex→ dos tuberías

FIFOS→ es un fichero de tipo especial S_IFIFO

- Se crean mediante llamadas mkfifo() ó mknod
- Permite comunicación entre **dos procesos cualesquiera** sin estar relacionados
    - Cada proceso abre la fifo como quiere y realiza las op necesarias
- **Persistentes**→ sobreviven al proceso que las crea

Procesos y FIFOs

- **FIFO**→ debe tener un proceso lector y uno escritor, hay que tener en cuenta el solapamiento en las escrituras
- **PIPE_BUF**→ define el nº máximo de caracteres que se pueden escribir en una FIFO **atómicamente**
- Funcionamiento depende del flag o O_NONBLOCK:
    - Si no hay un open() de solo lectura se bloquea hasta que otro proceso abra el FIFO para escritura y viceversa
    - Si hay un open() de solo lectura retorna inmediatamente, si es de solo escritura el open() y ninguna FIFO abierta para lectura retorna error

Crear una FIFO y abrirla para lectura y escritura involucra **tres llamadas al sistema,** con pipe() 1 sola

Llamadas bloqueantes:

- Un proceso abre una FIFO en lectura sin existir procesos escritores
- Un proceso abre una FIFO solo escritura sin haber lectores

Modelos de E/S para transferencia de datos en canales punto a punto:

- **Chorros de bytes:**
    - Sin delimitadores
    - El receptor no sabe si los datos que recibe fueron escritos de una vez o muchas

- **Mensajes:**
    - **Estructura común**→ formato conocido por ambos, requiere acuerdo previo en forma de código compartido
    - **Delimitadores de fin de mensaje**→ acuerdo en el carácter delimitador

T5.2 SOCKETS

Es un **punto final de comunicación** (socket→ dir IP+puerto+protocolo)

Diseño independiente del : protocolo, lenguaje y SO

Es una abstracción que:

- Ofrece interfaz de acceso a servicios de red en nivel de transporte
- Representa un extremo de una **comunicación bidireccional** con una dir asociada

Sockets y puertos

Socket→ ligado a una pareja(dir IP+puerto) y un protocolo.
Socket= elemento del proceso
Puerto= elemento del SO→2^16 disponibles
No se puede reabrir un puerto ya asignado a otro proceso

Servidor→ proceso que se ejecuta en un nodo de red y proporciona un determinado recurso o servicio:

- **Interactivo**→ recibe y atiende la petición, grandes esperas si atiende lento
- **Concurrente**→ recibe las peticiones y genera un proceso por cada una para atenderlas

Cliente→ proceso ejecutado en el otro nodo que realiza peticiones al servidor, inicia y termina el diálogo

Implementación cliente servidor

Servidor:

1. Creación y enlace al puerto de servicio de un socket
2. Desconexión del servidor de su terminal de lanzamiento o ejecución

3. Bucle infinito en el cual el servidor:
    a. Espera petición
    b. La trata
    c. Calcula y formatea la respuesta
    d. Envía

Cliente:

1. Creación del socket local
2. Preparación de la dir del servidor
3. Envío del mensaje
4. Espera del resultado
5. Explotación del resultado

1.Dominios de comunicación→ representa una **familia de protocolos**

Socket asociado a un dominio desde que se **crea**, cada dominio tiene su formato de direcciones
Servicios de sockets son **independientes** del dominio, comunicación solo entre sockets del mismo dominio

2. Tipos o estilos de sockets→ recoge el conjunto de propiedades del servicio que se desean, independiente del dominio de comunicaciones

Stream→ al conectar se realiza una búsqueda de un camino libre entre origen y destino y se mantiene el camino en toda la conexión.
PF_INET→ TCP PF_UNIX= FIFO full-duplex

- Orientado a **conexión**→ full-duplex
- **Fiable**→ no se pierden ni duplican
- **Secuencial**→ orden de entrega de los datos
- No hay separación entre mensajes
- Envío fuera de banda

Datagrama→ basada en datagramas no orientada a conexión, cada paquete podrá ir por cualquier sitio
PF_INET→ UDP

- Sin conexión
- No fiable
- No se garantiza recepción secuencial
- Separación entre mensajes

Otros:

Raw→ acceso a protocolos de niveles más bajos, requiere uso de root y se usan para nuevos protocolos de transporte (RDP)
Sequenced packet sockets→ presentes en PF_NS, = a stream pero límites de mensajes
Reliably delivered message sockets→ garantizan la entrega supuestamente

3.Direcciones de sockets→ debe tener asignado una dirección única

- Dependientes del dominio
- Usos:
    - Dirección local a un socket- **bind()**
    - Especificar una dirección remota -connect(), sendto()
- Formatos de direcciones: cada dominio usa una estructura específica:
    - PF_INET→ dir IP + puerto → socket.AF_INET
        - IP del host:32/128 bits
        - Puerto : 16 bits
        - Transmisión= host + puerto origen y destino y protocolo de transporte
        - Al pasar dir a la pila TCP/IP han de codificarse en el orden de bytes de red
    - PF_UNIX→ path del fichero

4.Mapa de la API de sockets:

1. **Creación**→ socket():creas bind(): asignas dirección
2. **Preparación de la conexión**
    a. listen(): esperar conexiones en modo pasivo
    b. connect(): iniciar conexión a otro socket
    c. accept(): aceptar una conexión
3. **Transferencia de datos:**
    a. write/send/sendto→ escribir datos
    b. read/recv/recvfrom→ leer datos
4. **Despedida y cierre**→ close()/shutdown: cerrar un socket eliminando la conexión

Semántica de aceptación de conexión:

- Solo se usa en el lado del servidor(en tipo stream), extrae la primera petición de la cola creada con listen()
- Cuando se produce la conexión el servidor obtiene como retorno:
    - Dirección del socket remoto cliente
    - Un nuevo descriptor que queda conectado al socket del cliente cuya petición se ha extraído
- Al retornar accept() quedan dos sockets activos en el servidor:
    - Original para las conexiones
    - Nuevo para enviar/recibir datos por la conexión establecida
- Gracias a esto se pueden hacer servidores multiprocesos para servicios concurrentes

Envío y recepción de datos

Envío:

- llamadas al sistema para **escribir datos** en un socket: write, send , sendto

- Write() = que con ficheros (os)
    - Se usa el descriptor del socket
    - Puede escribir en cualquier descriptor
    - Devuelve el nº bytes enviados

Recepción:

- Read = qué con ficheros (os)
    - Puede leer de cualquiera socket o fichero
    - Con flags=0 recv()=read en ficheros

No es necesario **establecer conexión** para transferir basta con crear y reservar dirección bind()

Socketpair()--> estructura similar a pipe, pero en esta ambos sockets son bidireccionales. Pareja de sockets sin nombre y conectados

5.Gestión de direcciones

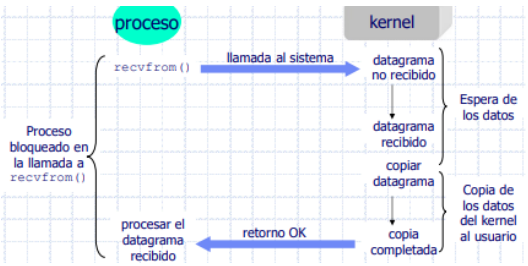
- Direccionamiento en sockets
    - Los usuarios manejan direcciones como textos
    - Las llamadas de sockets manejan direcciones en binario
    - Conversión entre ambos formatos:
        - Direccionamiento físico → decimal- punto a binario , binario a decimal-punto
        - Direccionamiento lógico→ dominio-punto a binario, binario a dominio-punto
- Orden de los bytes→ **orden de red,** big endian
- Otras funciones
    - Nombre de host local
    - Dir local y remota de un socket

JUEVES NOCHE

T6 - Técnicas avanzadas de programación en Red

E/S Avanzada:
Read(): Dos fases, esperar a que los datos estén listos → Copiarlos del kernel a la mem del proceso.
Write(): Esperar al destino que esté ready para recibir los datos → Escribir los datos de la memoria del proceso a la memoria del kernel

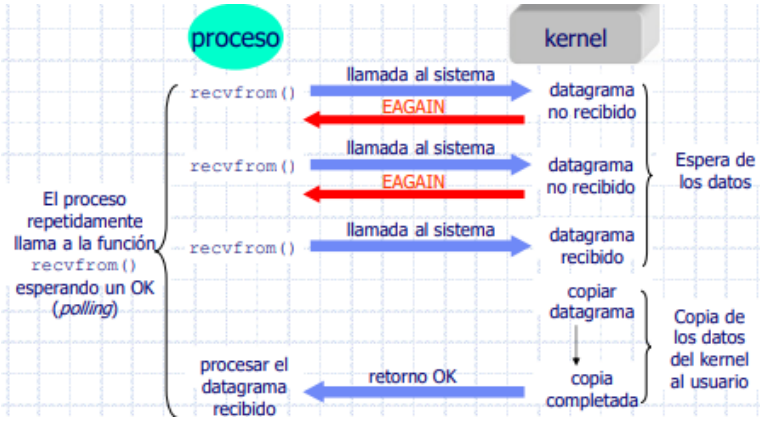
E/S Bloqueante: El proceso espera (queda **bloqueado)** a que la lectura/escritura este ready
Contras: ¿Cómo hago lectura/escritura a la vez (full-duplex)? ¿Hacer algo mientras espero a que termine E/S? ¿Cómo leo más de una fuente de datos a la vez?



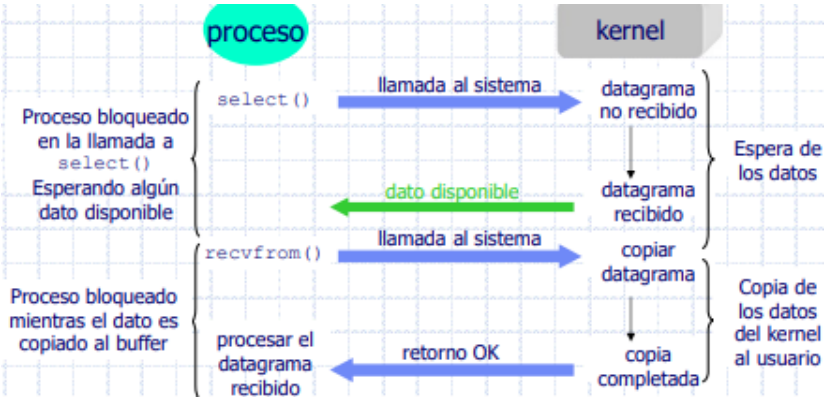
E/S no Bloqueante: Sirve para evitar que cuando E/S vaya a bloquear el proceso,no lo bloquee, lo **notifica** como un **error**

Polling: Se pregunta constantemente al server si está ready para realizar la operación. El tiempo de pregunta debe ser **pequeño** para evitar tardar, pero **grande** para evitar saturar el PC

- Es un desperdicio de CPU



E/S Multiplexada: Construye una lista de descriptores en los que estamos interesados. Llama a una función que no vuelve hasta que uno de los descriptores esté **listo para E/S**



E/S Conducida por señales:

El kernel nos notifica con alguna señal cuando una operación de E/S (por la que un proceso esperaba) está **lista para ser realizada**

Ventajas: Sistema no está bloqueado mientras espera que los datos estén listos

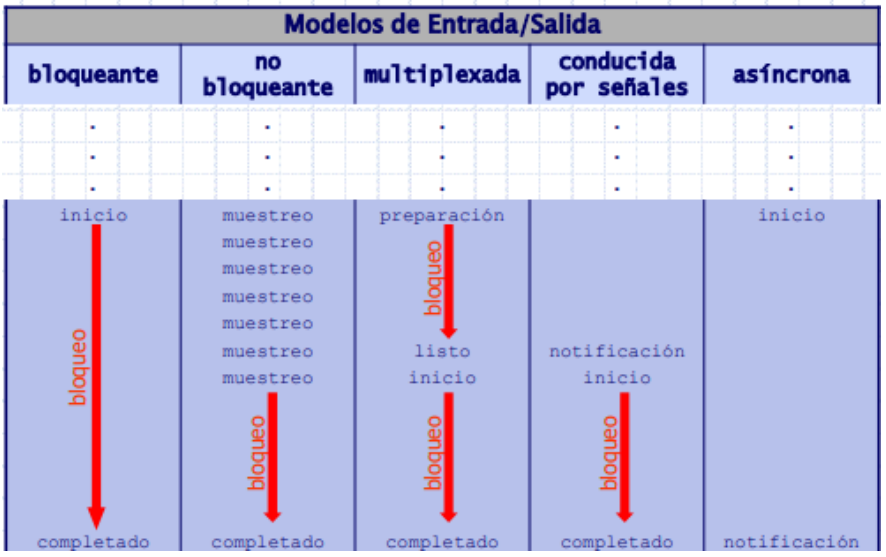
- Programa puede continuar ejecución hasta que se le notifique que el dato esté listo

E/S Asíncrona: Le indicamos al kernel el comienzo de una operación y esté nos avisa cuando se **complete entera**, incluyendo el copiado de datos a la memoria del proceso desde el kernel

*En el modelo de señales el **kernel** nos avisa cuando la operación **E/S** puede ser iniciada
*En el asíncrono el kernel nos indica cuando la operación está completada

aio_read() le pasa al kernel el **descriptor** de E/S, un puntero al **buffer** intermedio**,** el **tamaño del buffer** y un **desplazamiento de fichero**.

- Esta llamada se retorna inmediatamente y el sistema no se bloquea por E/S



E/S Multiplexada

Se realiza mediante **select()** → Se le pasa una lista de descriptores. La llamada no devolverá el control hasta que uno de los descriptores de los dispositivos E/S esté listo. La llamada nos **devolverá** los descriptores listos para su uso

```
import select
ready_rlist, ready_wlist, ready_xlist =
      select.select(rlist, wlist, xlist[, timeout])
```

En Unix np, En Win solo funciona con sockets
Timeout: Si es igual a 0 → Como un polling // Timeout = 4.5 //Timeout = None

¿Cuando está listo un descriptor?
rlist → Cuando es seguro que una lectura de dicho descriptor **no se bloqueará**
wlist → Cuando es seguro que una escritura sobre dicho descriptor **no se bloqueará**
xlist → Se reciben datos **fuera de banda** en red. En determinadas condiciones en pseudo-cli

Poll(): Más eficiente que la llamada **select()**:
Poll: O(Nº fd registrados)) Select(): O(Nº de descriptor más alto)

```
import select
objeto_poll = select.poll()
```

T7-UNICODE

Carácter→ el componente más pequeño del lenguaje escrito con valor semántico
Glifo→ representa la **forma** del carácter desplegado, los contienen las fuentes

A latina y A griega son→ **distintos caracteres = glifo**

Las tablas generadas por Unicode dan **propiedades** al carácter: letras, nº, símbolos…

Composición dinámica

- No se puede representar la Q con circunflejo, sino con U+0302 circunflejo combinado, que no es lo mismo que el ^ de ASCII
- Las fuentes pueden tener un glifo precompuesto para la Q con ^

Texto plano:

- La info suficiente para simplemente leerse
- Es público, estandarizado y legible universalmente (SGML,HTML,XML)

Orden lógico→ los caracteres son representados en orden lógico , unicode proporciona un algoritmo de tabla dirigida para reordenar el texto en el orden de lectura adecuado incluyendo direcciones mixtas

Unificación→ si los carácteres se ven iguales y provienen de diferentes estándares de origen, son un único carácter Unicode

- Signos de puntuación, símbolos,etc están unificados
- Diferencias en lenguaje,fuente,tamaño y posición no están representadas

- Caracteres que parecen el mismo de diferentes scripts no están unificados

Equivalencia→ maneras distintas de representar los mismos caracteres son válidas

Convertibilidad→ los caracteres de otros conjuntos se pueden convertir 1:1 en unicode, se realizan por tablas de mapeo

Codificaciones

ASCII→ 7 bits codificados para 100 caracteres

ISO-8859-1→ 8 bits para 200 caracteres

Shift-JIS→ 8/16 bit para 8k caracteres

3 Codificaciones unicode→ **UTF:** algoritmos de mapeo de de cada punto de código Unicode a una única secuencia de bytes, los 3 pueden ser transformados entre ellos sin pérdida de datos (UTF-8,16,32)

UTF-8→ HTML y similares

- Manera de transformar todos los carácteres unicode en una longitud variable de codificación de bytes
- No tiene problemas de ordenamiento, la transformación a UTF-8 puede usarse sin modificación entre mucho software existente

UTF-16→ popular en entornos que balancean **acceso eficiente** a caracteres con uso económico de **almacenamiento**

- **Compacto**
- Cada carácter BMP representado por 16 bits otros, representados por 2 de 16-bits

UTF-32→ útil cuando no preocupa la memoria,pero el ancho es fijo

- Los puntos de código Unicode están directamente indexados, pero es ineficiente con el espacio ya que usa 4 bytes para cada punto de código

---

VIERNES MAÑANA

SIMULACRO CON TEST

CLASE TEST

```
import unittest;
import time

from serv_bueno import Servidor

class TestServidor(unittest.TestCase):

    def test_fecha(self):
        servidor = Servidor()
        fecha = time.strftime("%A %B %d %Y")
        parametro = "fecha"
        fecha2 = servidor.fecha(parametro)
        self.assertEqual(fecha2,fecha)

    def test_hora(self):
        servidor = Servidor()
        hora = time.strftime("%H:%M:%S")
        parametro = "hora"
        hora2 = servidor.hora(parametro)
        self.assertEqual(hora2,hora)

    def test_a_tomar_por_culo_el_strip(self):
        servidor = Servidor()
        parametro = "fecha "
        fecha = servidor.fecha(parametro)
        self.assertEqual(fecha,"Error en el formato")
```

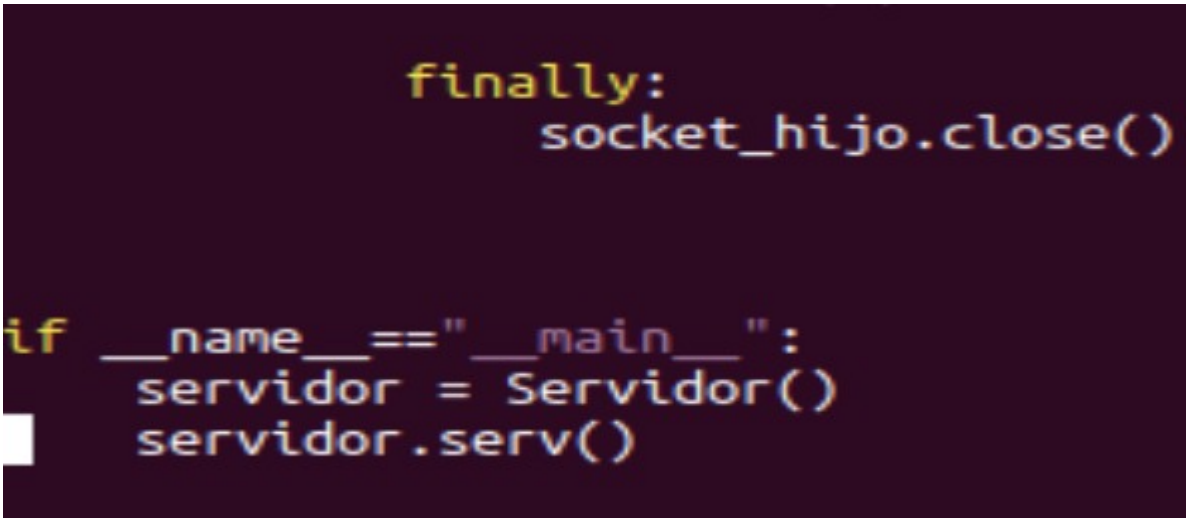

CLASE SERVIDOR

```
class Servidor:
    def fecha(self,parametro):
        if parametro == "fecha":
            fecha = time.strftime("%A %B %d %Y")
            return fecha
        else:
            fallo = "Error en el formato"
            return fallo

    def hora(self,parametro):
        if parametro == "hora":
            hora = time.strftime("%H:%M:%S")
            return hora
        else:
            fallo = "Error en el formato"
            return fallo

    def minusculas(self,peticion):
        peticion_minusculas = peticion.lower()
        return peticion_minusculas
```


DEFINICIÓN DE LAS FUNCIONES

```
datos = socket_hijo.recv(1024)#recibe el nombre fichero

if not datos:# ctrl c
    break
datos = self.minusculas(datos.decode("utf8"))
```

LLAMADA A LAS FUNCIONES



FINALIZACIÓN

OTRA MANERA SEPARANDO FUNCIONES

CLIENTE

```
import socket

s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

address = './uds_socket'

s.connect(address)

mensaje = input("FECHA/HORA: ")
mensaje = mensaje.lower()

s.send(mensaje.encode('utf8'))

mensaje = s.recv(4096)
```

```
print(mensaje.decode('utf8'))


mensaje = input("FECHA/HORA: ")
mensaje = mensaje.lower()

s.send(mensaje.encode('utf8'))

mensaje = s.recv(4096)
print(mensaje.decode('utf8'))


mensaje = input("FECHA/HORA: ")
mensaje = mensaje.lower()

s.send(mensaje.encode('utf8'))

mensaje = s.recv(4096)
print(mensaje.decode('utf8'))
```

SERVIDOR

```
import socket, time, os
from metodos import Funciones

print("Esperando peticiones")
address = './uds_socket'

try:
    os.unlink(address)
except OSError:
    if os.path.exists(address):
        raise

funciones = Funciones()
s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
s.bind(address)
s.listen(1)

class Metodo:
    def servidor():
        con, client_addr = s.accept()

        try:
            for i in range(3):
                mensaje = con.recv(4096).decode('utf8')
                if not mensaje:
                    break
                if mensaje == "fecha":
                    resultado = funciones.Fecha()
                    con.send(resultado.encode('utf8'))
```

```
            elif mensaje == "hora":
                resultado = funciones.Hora()
                con.send(resultado.encode('utf8'))

            else:
                resultado = funciones.Error()
                con.send(resultado.encode('utf8'))
        finally:
            con.close()
    pid = os.fork()
    if pid == 0:
        servidor()
    else:
        servidor()
    while True:
        servidor()
```

TEST

```
import unittest, time

from serv import Funciones

class Tests(unittest.TestCase):
    funciones = Funciones()
    def test_something(self):
        fecha = time.strftime("%Y-%m-%d")
        fecha_calculada = self.funciones.Fecha()
        self.assertEqual(fecha, fecha_calculada)  # add assertion here


if __name__ == '__main__':
    unittest.main()
```

FUNCIONES

```
import time
class Funciones:
    def Fecha(self):
        fecha = time.strftime("%Y-%m-%d")
        return fecha

    def Hora(self):
        hora = time.strftime("%H:%M:%S")
        return hora

    def Error(self):
        resultado = "ERROR"
        return resultado
```