

# CLASES TEORIA

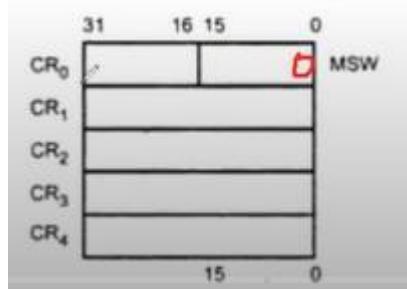
Cuando el procesador **comienza a arrancar** el ordenador, se inicia en **modo real** y funciona como un procesador 8086. El procesador puede ver hasta 1 MB de RAM.

El procesador cambia al **modo protegido** mientras carga Windows \* u otro sistema operativo avanzado. En el modo protegido, el procesador utiliza direccionamiento segmentado (no lineal), en lugar de direccionamiento lineal.

El **direccionamiento segmentado** significa que la memoria (memoria física y virtual) se divide en los bloques 64K. Este es el valor máximo para el registro del puntero de instrucción (IP). El registro IP funciona con el registro del segmento de código (CS) para apuntar a la ubicación de memoria desde donde el microprocesador debe buscar su siguiente instrucción. El IP utiliza 4 bytes para direccionamiento de memoria, haciendo 0FFFFH la ubicación de memoria máxima ( $0FFFFH = 64K$ ).

## Clase 1: Modo Protegido

Dentro de los registros internos del procesador: en el **CR0**:

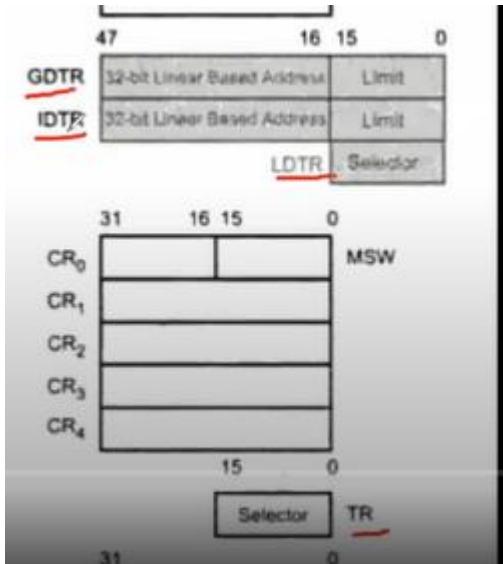


Si lo **ponemos en 1**, pasamos al modo protegido: (lo hace el SO o podemos hacer una app q lo haga)

Donde se tienen todas las funcionalidades

Puedo direccionar 4GB de RAM

Puedo acceder a los registros:



**GDTR:** Registros de la tabla de descriptores globales

**IDTR:** Registro de la tabla de interrupciones

**LDTR:** Registro de la tabla de descriptores locales

**TR:** Registro de tareas (de cambio de tareas).

Los registros **CR0, CR1, CR2, CR3** y **CR4** son registros de control que se utilizan para controlar y configurar el funcionamiento del procesador.

#### Protección / Aislación (entre tareas):

Si tengo más de uno corriendo, es imposible q una tarea **lea o escriba** información de otro.

Si intento acceder a una posición de memo q no es mía, da una excepción de hardware.

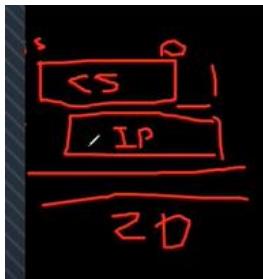
#### Hay 4 niveles de privilegio:

El mas privilegiado: tiene acceso a todo el set de instrucciones (instrucciones E/S también). Los **SO** trabajan en este. **Modo Privilegiado o Modo Kernel**.

El ultimo nivel, el menos privilegiado, es como un set de instrucciones recortado. Cuando escribo un programa, trabaja en este. **Modo Usuario**.

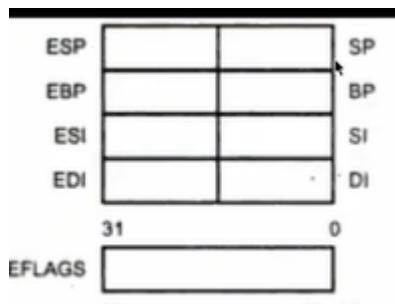
#### Como se direcciona la memoria:

En **modo Real**, para buscar una instrucción, se desplazaba el registro 4 bits y se le suma el offset:



Se obtenía la **dirección física**.

Los desplazamientos (offset) son de 32 bits, a diferencia q en modo real.



El Stack Pointer ahora se llama **Extended Stack Pointer (32 bits)**, así como los demás...

En **modo protegido**:

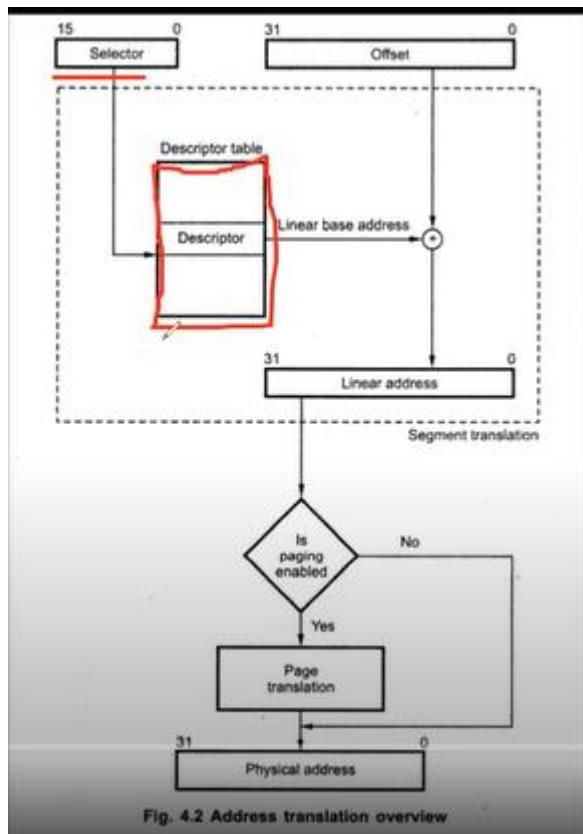
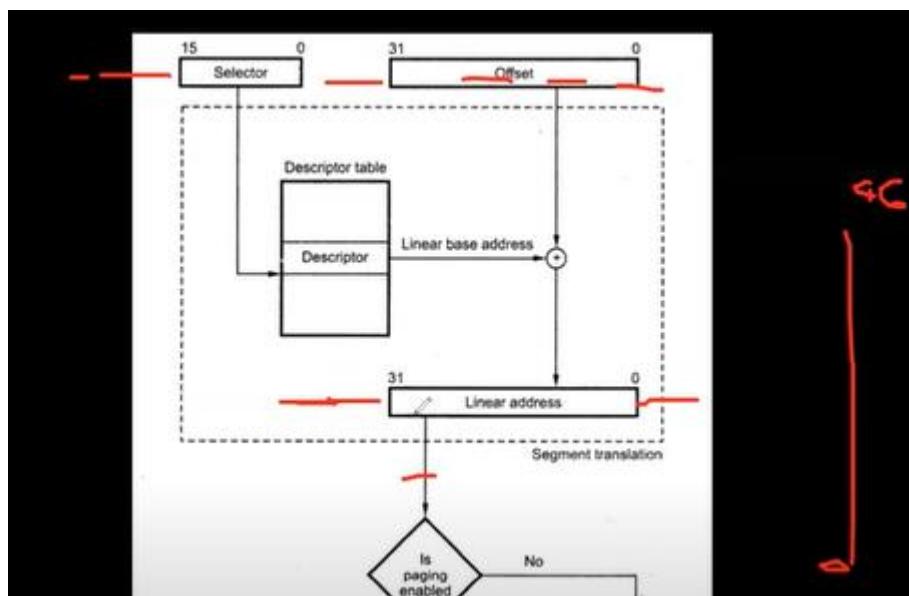


Fig. 4.2 Address translation overview

La **tabla de descriptores**, esta en la memoria principal (RAM). Los **descriptores** tienen dentro: Dirección base, un límite (algunos), bits de privilegio.

Lo primero q se hace es un proceso de **Traslación** llamado **Segmentación**, si o si ocurre **esta traslación de direcciones**. Se parte primeramente de una dirección virtual, luego de la traslación el procesador ve una memoria lineal de 4GB:



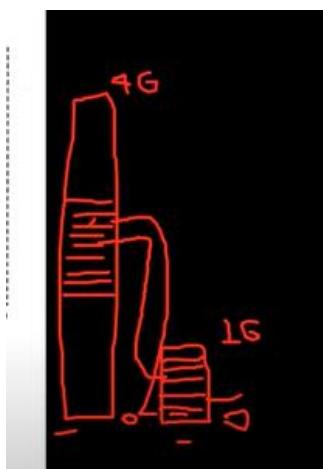
Si tengo algunos bits de los registros de control habilitados, se hace una segunda traslación: **Paginación**.

Si no existe esta segunda, la dirección lineal equivale a la **dirección física** de la memoria.

La paginación sirve cuando por ejemplo se necesita ejecutar un proceso extenso q ocuparía mucha RAM, lo q hace es fraccionarlo en varios, dejando algunos fragmentos q no se utilizan tanto fuera de la RAM, y dejando dentro los q si se están utilizando en algunos espacios de memoria designados. Aprovechando así mejor la memoria física.

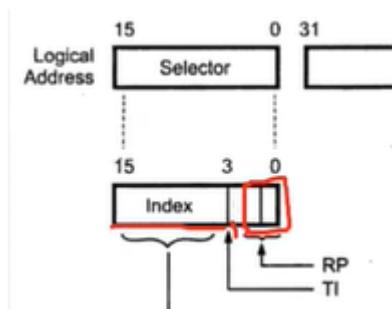
El que accede a la memoria principal para mover paginas lo hace el SO.

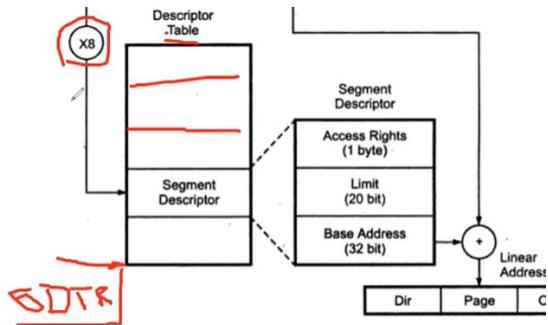
Ejemplo del profe: La de 4GB es la memoria lineal q ve el procesador, la de 1GB es la memoria física real de la RAM, ambas divididas en 4KB. En la de 4GB cada subdivisión se llama **Paginas**, y en la de 1GB se llaman **Marcos de Pagina**.



El selector tiene (16bits):

2 bits q tienen q ver con el nivel de privilegio, 1 bit q dice q tabla de descriptores es (**local** = 1 y **global** = 0), y el resto sirve para moverse como offset dentro de la tabla de descriptores.





El **x8** esta porq la tabla de descriptores es 8 bytes (64 bits).

Hay una sola **tabla global** en el sistema, pero puede haber una o más **tablas locales**. Si es global accedo a través del registro **GDTR**.

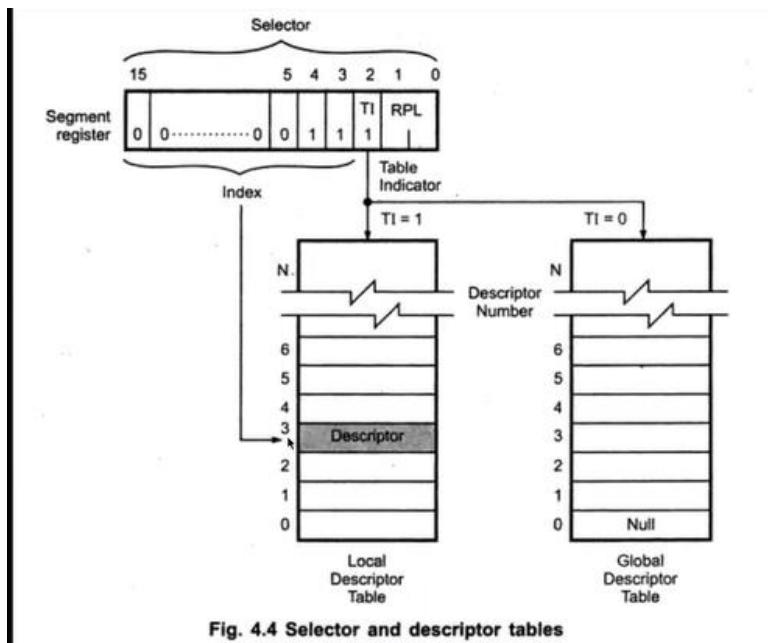
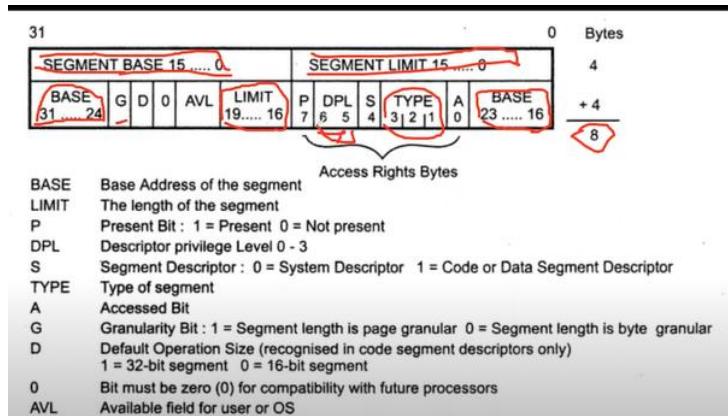


Fig. 4.4 Selector and descriptor tables

Ejemplo de un segmento, pero lo importante es la **base**, el **límite** y los **permisos**:



Cada tarea va a tener acceso siempre a la tabla global q es única en el sistema, pero también podía tener acceso a una tabla local para esa tarea:

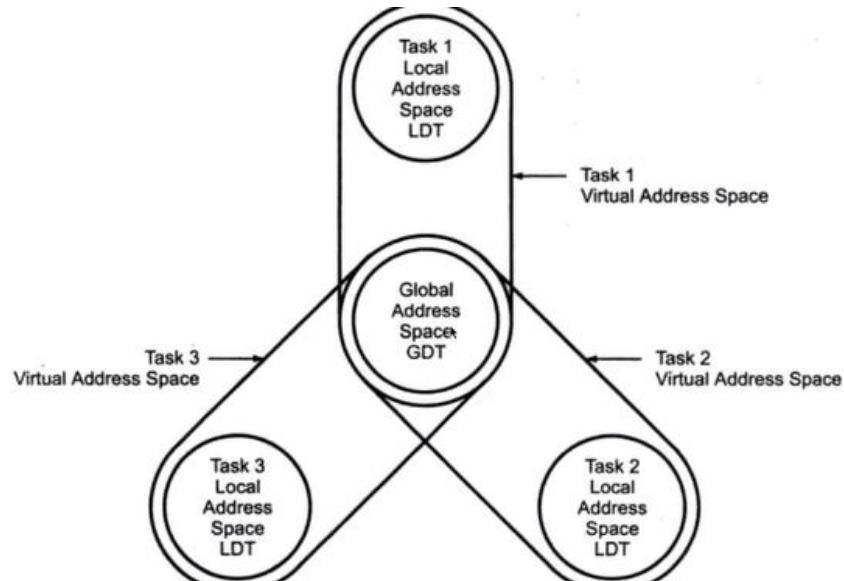
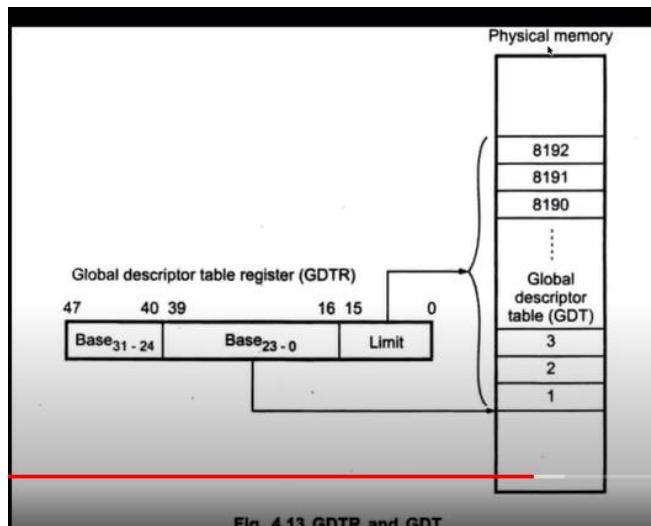


Fig. 4.12 Memory area shared by different tasks

Para acceder a una tabla local, primero tengo que acceder a la global.

Para buscar un descriptor en la tabla global el procedimiento es:



Se utiliza del GDTR la base para localizar desde donde empieza la tabla global en la memoria física, y de ahí con el index (indice) del registro selector, busco q descriptor necesito.

Para buscar un descriptor en la tabla local, es un poco distinto:

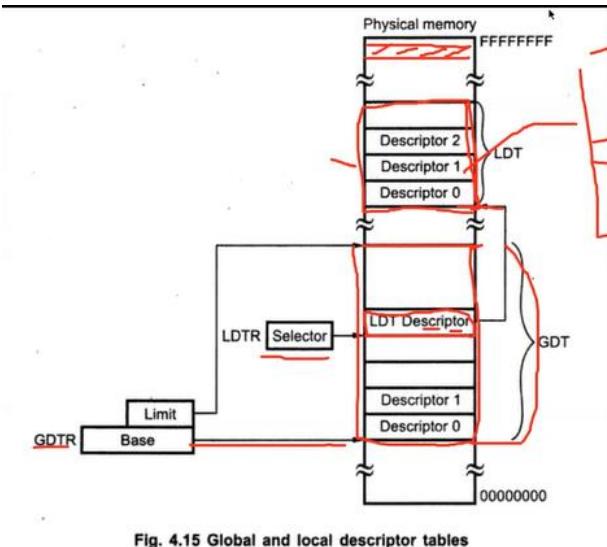


Fig. 4.15 Global and local descriptor tables

Primero busco la tabla global con la base del GDTR, luego utilizo el registro LDTR como offset para localizar un segmento LDT que tiene info de la base y el offset de una tabla de directores locales, y uno de esos descriptores dentro de la LDT mapea a otro área de la memoria principal.

(LDTR = 16 bits y GDTR = 32 bits mas los 16 q me dicen el offset)

Hasta acá seria segmentación, si yo no tengo paginación hasta acá seria la dirección lineal, es decir la dirección física.

## Paginación:

La memoria virtual tiene **paginas** (bloques de 4KB) y la memoria física donde se van a alojar esas paginas tiene **marcos de pagina**.

La dirección lineal, se separa en 3 partes (10, 10 y 12 bits):

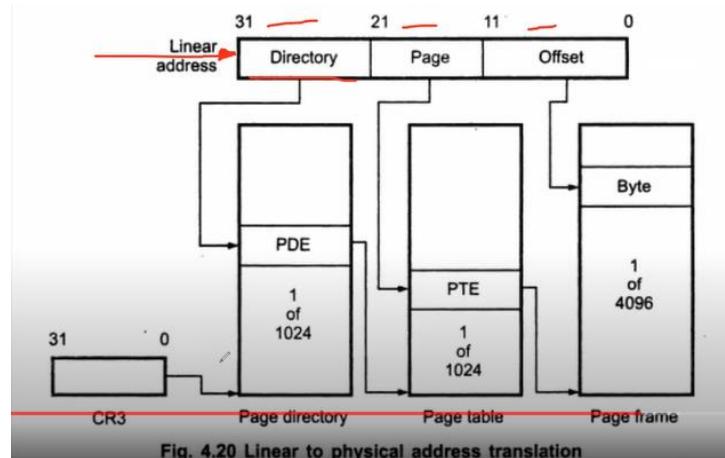
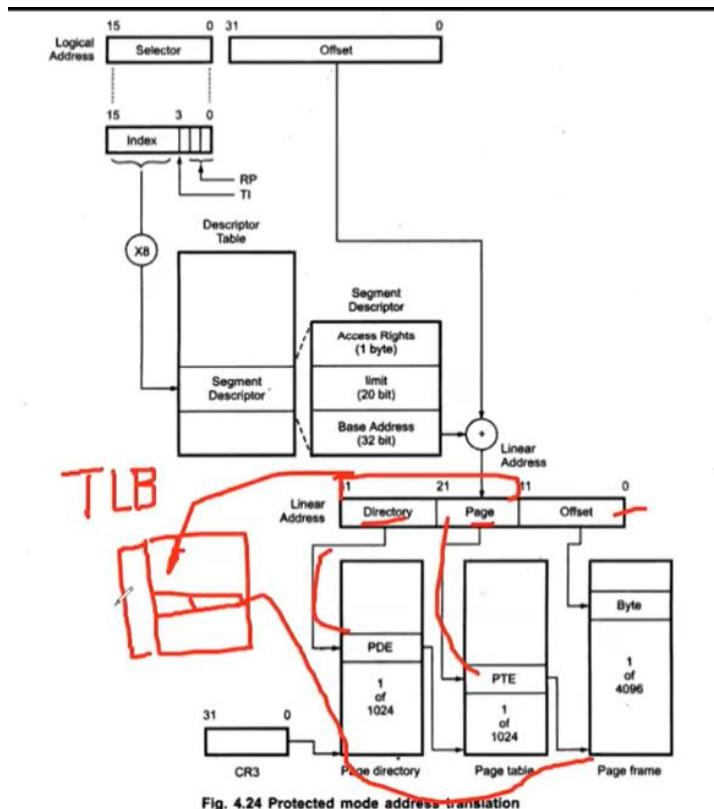


Fig. 4.20 Linear to physical address translation

El directory tiene info de un **PDE** dentro del **directorio de página** (1 entrada entre 1024 posibles). Esta tabla también esta en la memoria principal.

Dicho PDE tiene info sobre donde arranca la **tabla de página**, y con la parte Page encuentro el PTE necesario.

Y ese **PTE** tiene info sobre donde arranca la **Page Frame**, y con la parte Offset encuentro el byte necesario.



Como en total son 4 accesos a memoria principal, se creo el cache **TLB** para evitar acceder varias veces en la paginación y si existe coincidencia con la anterior instrucción entonces se evita acceder esas veces en la paginación.

En resumen, el objetivo de la memoria virtual es hacer creer al procesador q tiene espacios o direcciones distintas q nunca se van a chocar o interferir.

Esta traslación se va a una memoria lineal. Y si necesito de mayor memoria disponible, se parte como en pedacitos la tarea, es la función q cumple la paginación. Por lo q se tiene en memoria las partes q se están usando más. Si se llega a necesitar otra parte se carga en la memoria y así.

## Clase 2: Mecanismos de Protección IA-32

Protección contra programas q no se hagan lio entre ellos.

Existen 3 mecanismos de protección a nivel de segmentación y un par de mecanismos en paginación.

### Segmentación:

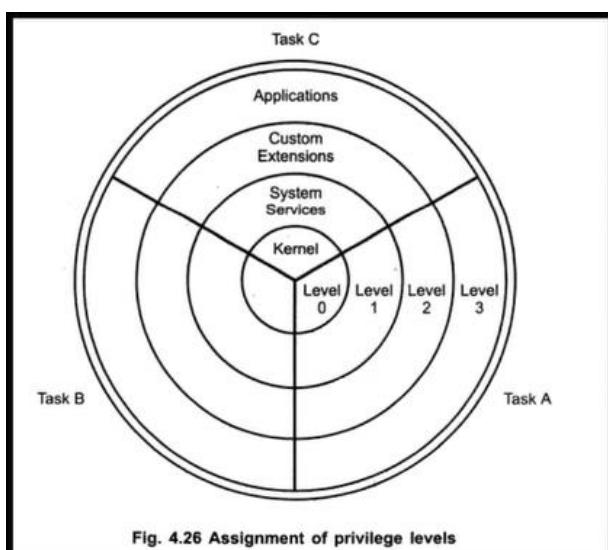
- **Chequeo de límite:** si la base mas el offset es mayor al limite, se tira una excepción.
- **Segundo mecanismo de protección:**

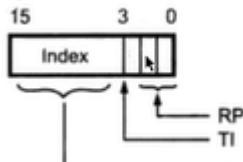
Table 3-1. Code- and Data-Segment Types

Decimal	Type Field				Descriptor Type	Description
	11	10 E	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	1	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

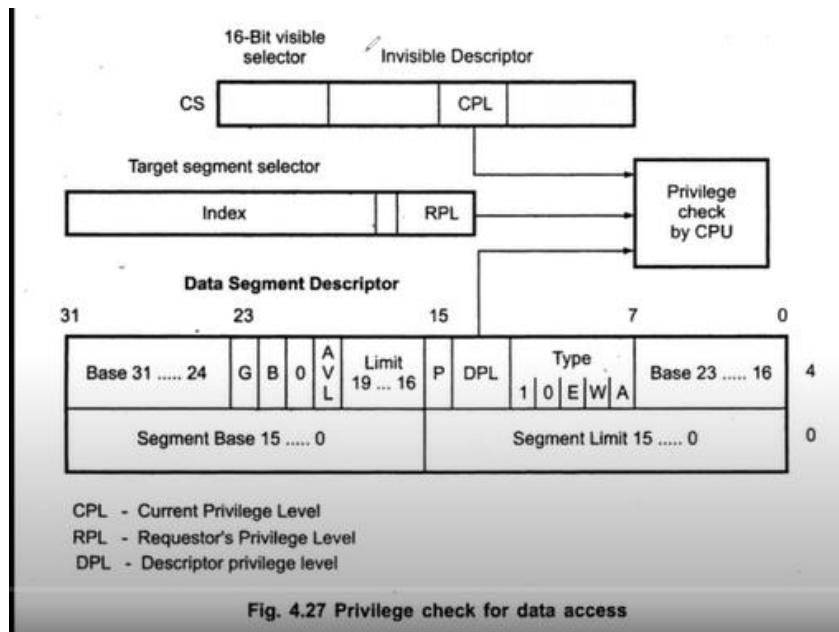
Si voy a acceder a un segmento de datos y el bit 11 esta en 1 entonces va a tirar una excepción.

- **Tercer mecanismo:**





Como tenemos dos bits de privilegio (RP) existen 4 niveles de privilegio.



El nivel q estoy ejecutando actualmente se llama **CPL** (current privilege level). Esta en la parte oculta de mi descriptor.

Si quiero ir a otro segmento de código, tengo q poner un valor nuevo en mi segmento de código: donde los últimos dos bits son de privilegio **RPL** (request privilege level, a donde yo quiero ir). Con los otros 13 bits (index) acceso al directorio determinado (el de abajo).

Este tiene 8 bytes, y posee 2 bits **DPL** q son de privilegio.

**El mayor entre el actual y el requerido debería ser menor o igual q el destino.**

(numéricamente)

$$\text{Max (CPL, RPL)} \leq \text{DPL}$$

(El valor numérico de privilegio mas bajo es el de mayor privilegio (El cero 0).)

Ejemplos:

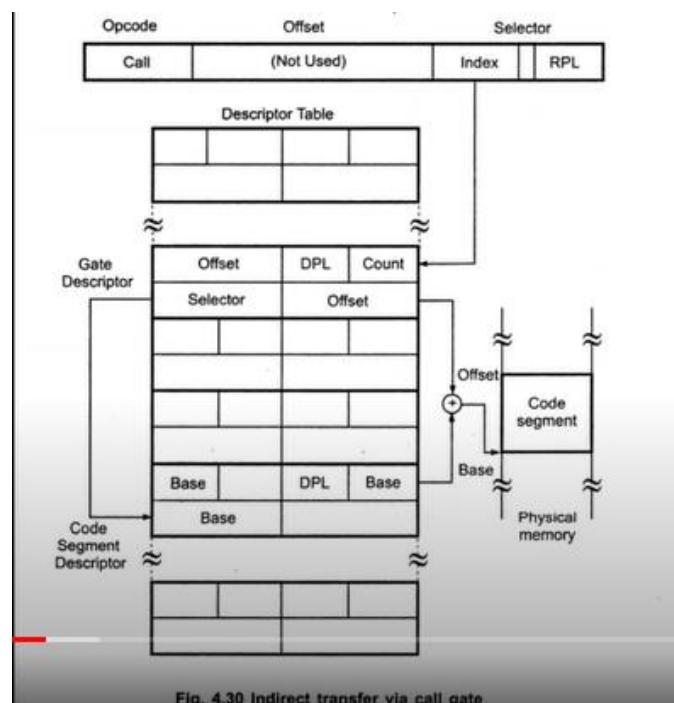
No	Privilege Levels			Access
	DPL	CPL	RPL	
1	2	0	1	Valid
2	3	1	2	Valid
3	1	1	0	Valid
4	1	2	0	Invalid
5	2	2	3	Invalid

Primer ejemplo, es valido porq estoy en el cero es decir en el mas privilegiado y quiero ir a otro de menor privilegio.

### CallGates:

Permiten ejecutar desde un nivel de privilegio menor, alguna petición con un nivel de privilegio mayor.

En vez de apuntar a un segmento en mi selector, al q no puedo ir, lo q hago es apuntar a un **descriptor de un callgate**.

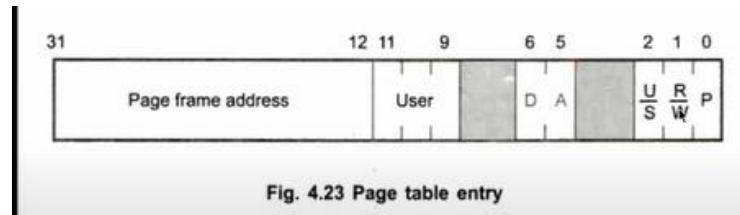


Con el índice buscamos un descriptor de puerta, y con el offset de este descriptor (no el q teníamos originalmente del selector), encontramos el segmento de código q buscábamos.

Es decir puedo ir a un segmento de código mas privilegiado, pero no con el offset q yo quiera si no el q me diga el descriptor de puerta.

## Mecanismos de privilegio en paginación:

Tanto el directorio de pagina como la entrada de la tabla de pagina, tiene los bits para limitar si es de escritura/lectura y de usuario/sistema:

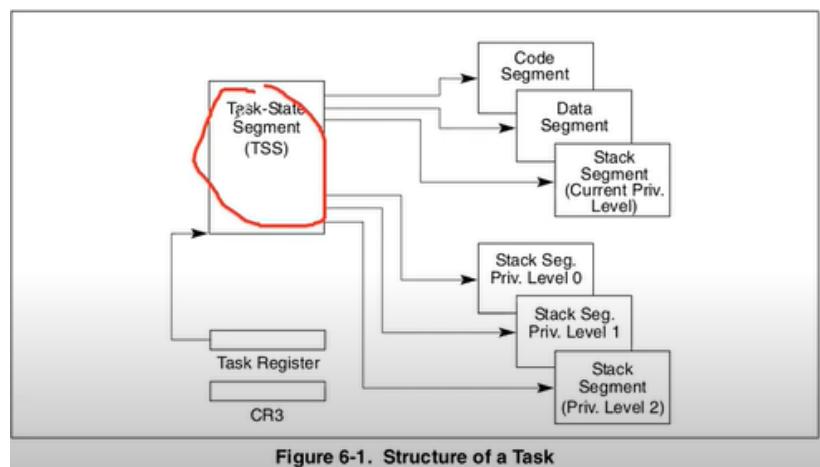


- Restricción del dominio direccionable:
- Chequeo de tipo:

## Multitasking:

Es necesario, en los casos por ejemplo de tiempos muertos en la ejecución de tareas. Como en la solicitud de entrada/salida, son tareas lentas q entre q se espera una respuesta, el procesador puede ser aprovechado mientras tanto con otras tareas.

En el **TSS** (task state segment), se van guardando los estados (foto actual) de las tareas:



Esto se hace porq la tarea no sabe en q momento se le va a sacar el uso de la CPU, por lo q constantemente se va guardando el estado actual.

Cada tarea va a tener un descriptor de task state segment, y si quiero cambiar de tarea, tengo q cargar en el **Task Register** el nuevo descriptor de tarea q quiero acceder.

Vista de q se guarda:

31	0
Bit Map Offset	0000000000000000 T
0000000000000000	LDT
0000000000000000	GS
0000000000000000	FS
0000000000000000	DS
0000000000000000	SS
0000000000000000	CS
0000000000000000	ES
	EDI
	ESI
	EBP
	ESP
	EBX
	EDX
	ECX
	EAX
	EFLAGS
	EIP
	CR3
0000000000000000	SS2
EIP	Editar título
0000000000000000	SS1
EIP1	
0000000000000000	SS0
EIP0	
0000000000000000	Back link

Fig. 5.1 Task state segment

Descriptor de segmento de estado de tarea:

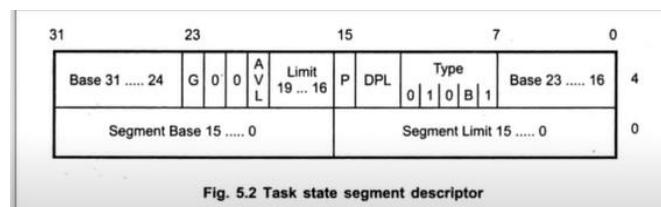


Fig. 5.2 Task state segment descriptor

El registro de tarea:

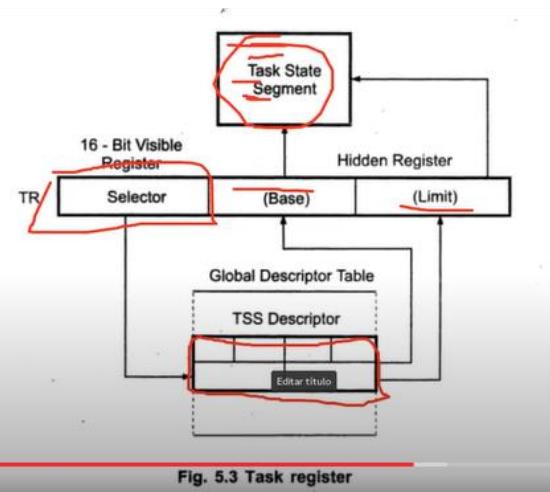
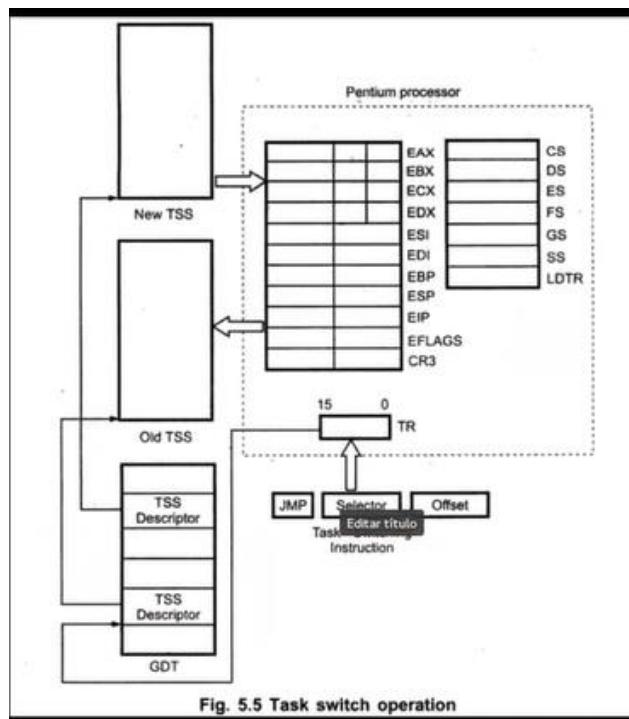


Fig. 5.3 Task register

Operación de cambio de tarea:



Hay 4 maneras de hacer el cambio, hay 2 q son con el jump **JMP** o con un **call**, pueden ser para ejecutar una entrada o salida, hay 1 q es cuando llega una interrupción se cambia la tarea y cuando termina una rutina de atención de interrupción (una **IRET**). Esas 2 son indirectas, con una **puerta de tarea**.

Cuando se vuelve de una **IRET**, la tarea q se tiene q seguir ejecutando luego de la interrupción esta guardada en el TSS, en el campo **back link**. Este back link se utiliza para estos casos nomas.

## Clase 3: Introducción a sistemas operativos

(minuto 38:20 aprox.)

Un sistema operativo es (2 enfoques):

- **Maquina extendida:** Brinda una interfaz genérica y única para lo q yo quiera hacer. De mas alto nivel para los dispositivos de bajo nivel. Capa de abstracción (uso de Syscalls x ejemplo)
- **Manejador de recursos:**
  - Procesos
  - Memoria
  - I/O
  - Sistemas de archivos
  - Multiplexación (compartir recursos en tiempo y espacio)

También se define como un programa q brinda un entorno para q se ejecuten otros programas.

**¿Qué hace, q controla o gestiona?**

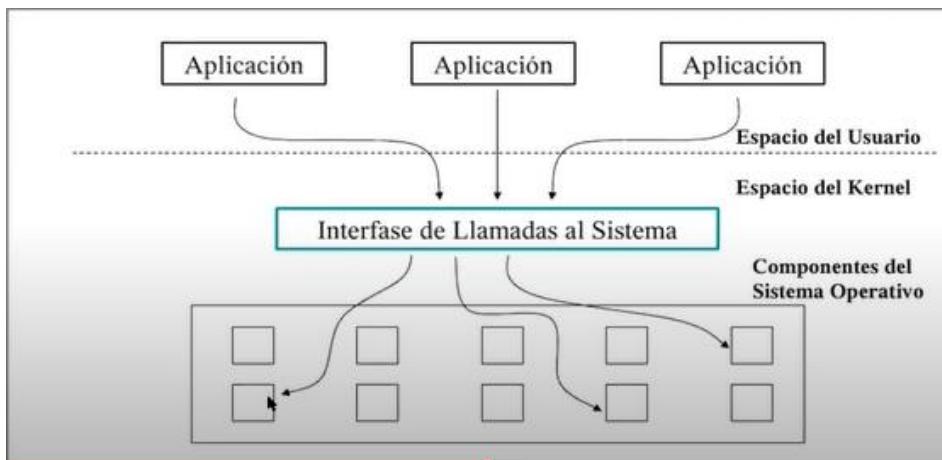
- **Gestión de procesos:** debería poder crear/terminar/suspender/reanudar procesos.
- **Gestión de memoria:** saber como esta distribuida la memoria, es decir q partes de memoria corresponden a cada proceso. Además también debe asignar/liberar memoria a los procesos.
- **Gestión de almacenamiento:** creación/borrado de archivos y directorios(carpetas).
- **Protección y seguridad:** evitar sobreescrituras, y monopolización de CPU. Necesita ayuda del hardware para esto (temas anteriores). Brindar permisos a usuarios (pero siempre en el menor nivel de privilegio).

**Tipos de SO:**

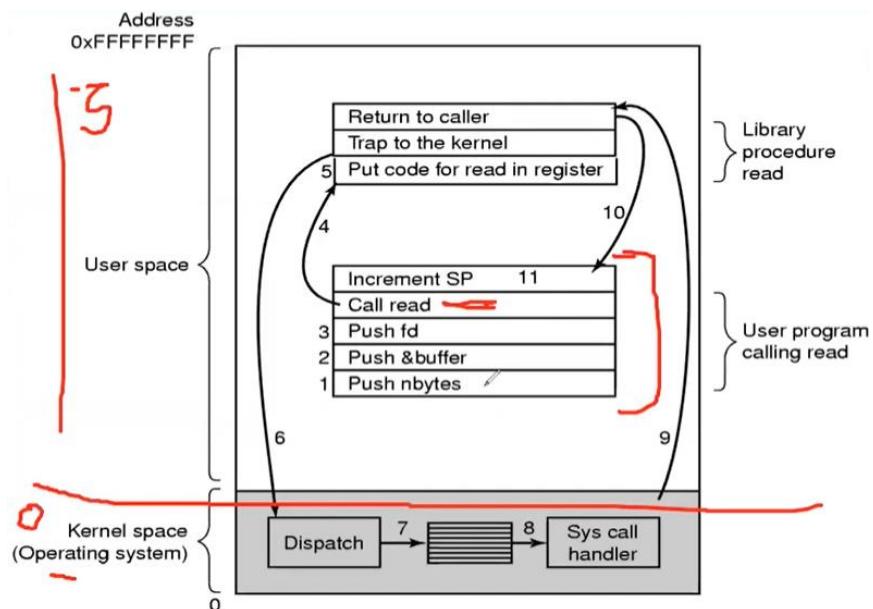
- **De propósito general:** para PC
- **De tiempo real:** tareas muy específicas.
- **Sistemas multimedia.**
- **Sistemas de mano (PDA)**
- **Sistemas centralizados**
- **Cliente servidor**
- **Peer to peer**
- **Basados en WEB**

## Estructura de un SO

El SO tiene componentes, en estos están los drivers o manejadores de controladores de HardWare. Por lo tanto si tengo una aplicación y necesito utilizar algún hard en específico, se hacen **llamadas al sistema** q estas harán uso de los componentes.



### Pasos de una llamada al sistema:

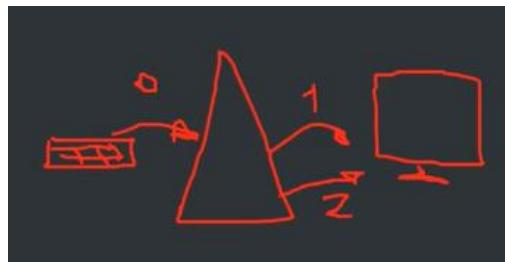


**Read** necesita 3 argumentos: 1ero se pone la **pila** (push nbytes) es decir la cantidad de bytes q voy a leer, necesita de un **buffer** (donde va a guardar lo q lee) y un **descriptor de archivos (fd)**.

Luego de la llamada de read, se pasa del modo usuario a modo Kernel. No a cualquiera lado ya q es un CallGate (apunta con el offset dado por esta). El programa se queda esperando hasta q el periferico entrega los datos al manejador de la llamada (handler), y

ahí recién se retorna al modo usuario. Luego se incrementa SP que lo q hace es desapilar lo q había en pila.

**Descriptores** (número entero asociado a un canal):

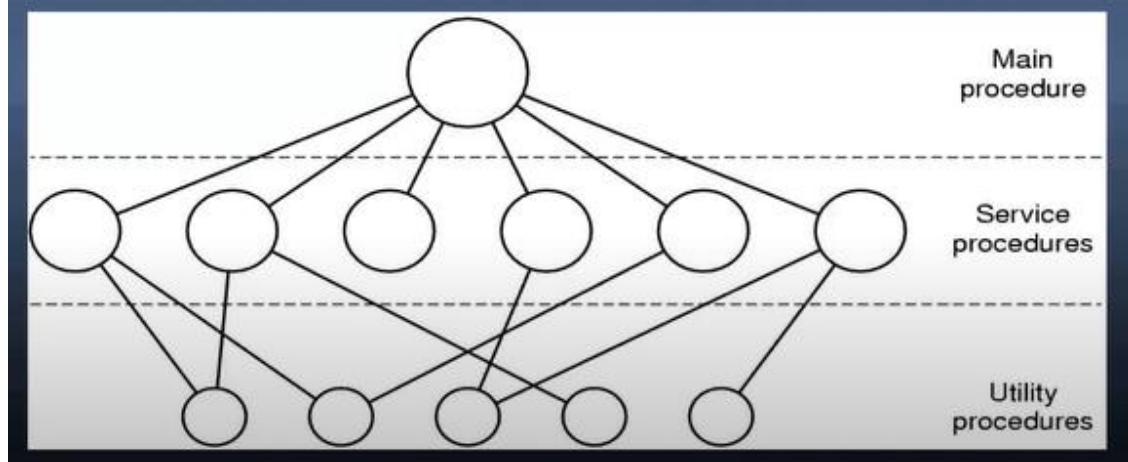


2: canal asociado a pantalla, pero para errores.

### Tipos de estructuras:

- **Monolítico:**

- **Monolítico**



Son todas funciones, se compilan y **es un solo binario q se está ejecutando**. Se puede organizar por servicios, donde hay una función principal después esta invoca los distintos servicios y también hay aplicaciones utilitarias. Es toda una sola cosa.

- ✓ Bien definida las interfaces entre las rutinas.
- ✓ Cualquier rutina puede llamar y ver a cualquier otra
- ✓ Componentes independientes entre si
- ✓ Soporte de extensiones: dll(Windows) y módulos.

**Pro:** desde cualquier servicio podría usar cualquier utilidad y desde el main podría invocar cualquier servicio.

**Contra:** tendría q tener bien en claro las interfaces porq podría hacer algo mal, es un solo un programa, falla uno falla todo.

- **En capas:**

- ✓ Capas jerárquicas:
- ✓ Cada capa se comunica con la adyacente
- ✓ Menor throughput q los monolíticos
- ✓ Es mas lento esto, porq va capa por capa

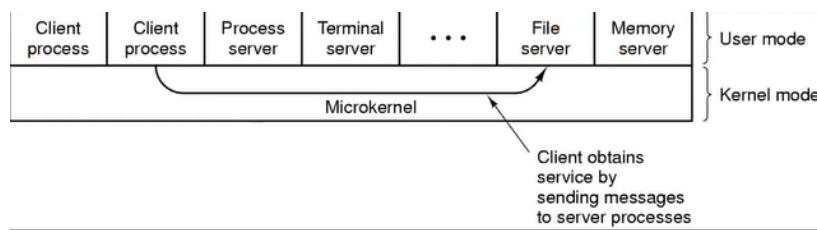
Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- **Microkernel:**

- ✓ Provee un reducido número de servicios
- ✓ Gran parte de las funciones del kernel se las lleva a capas superiores
- ✓ Microsoft (hibrido) y Minix

- **Cliente Servidor:**

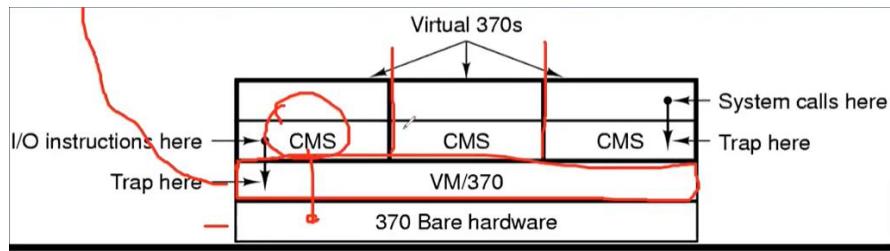
- ✓ Parecido al microkernel, pero con la diferencia q, los distintos procesos de modo usuario están en distintas máquinas (distintos clientes y servidores)



- **Máquinas virtuales:**

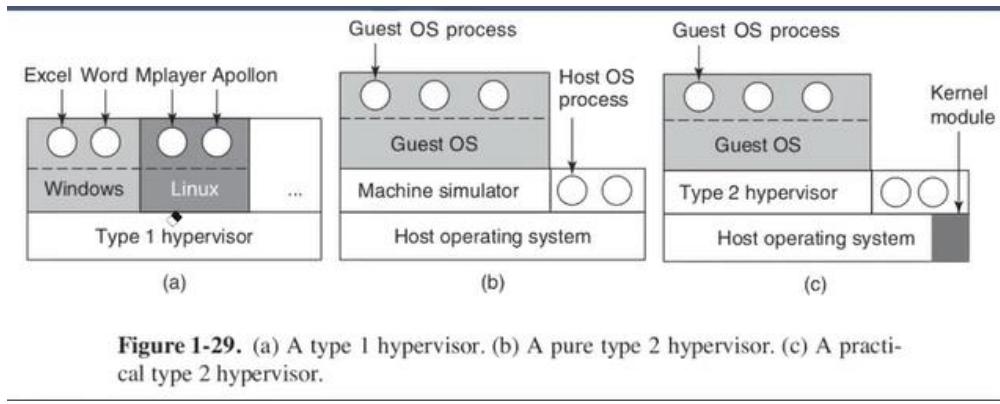
Se basa en la definición. Pero lo q hace es separar las funciones de: **multiprogramación** y **maquina extendida**.

Utilizan solo multiprogramación:



El VM sería el **hypervisor** q hace de comunicador con el hardware, y los CMS serían los SO instalados, q estos últimos creen q están comunicándose con el hardware directamente.

Se utilizan para ahorrar hardware, ya q el hypervisor lo multiplexa.



**Figure 1-29.** (a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor.

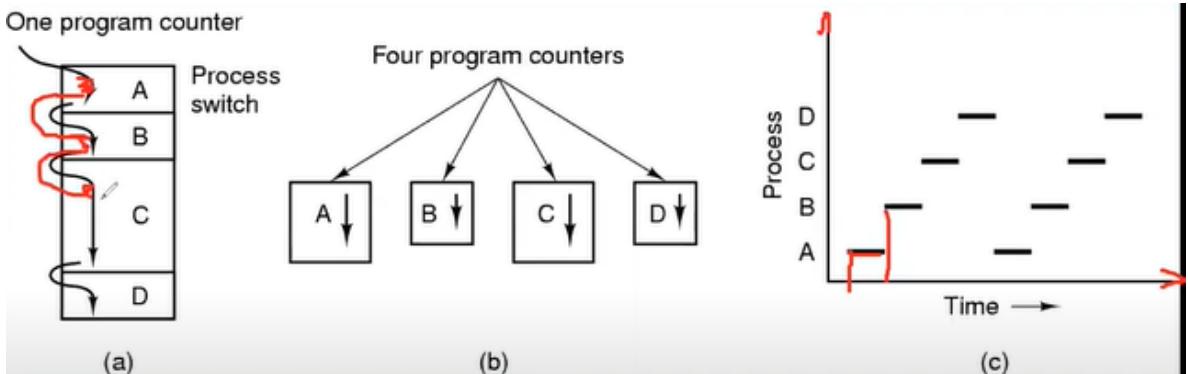
- a) No da maquina extendida
- b) Casi no se usa en la actualidad
- c) Se pone un SO con algunos módulos de kernel q manejan multiplexación de distintos clientes. El hipervisor es el q hace la gestión de los distintos clientes. (VirtualBox) El hipervisor no se comunica directamente con el hardware, pasa a través del SO.

## Clase 4: Procesos y Señales

### Procesos:

**Multiprocesamiento** → Para mayor velocidad del procesador y de entrada/salida.

**Pseudoparalelismo** → 1 solo procesador e instrucción a la vez. Los programas deben estar en memoria.



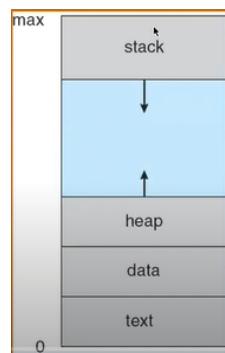
Se ejecutan secuencialmente (a) una parte de los programas por vez, veo como si se ejecutaran a la vez (b). Se va multiplexando en el tiempo la ejecución de cada uno de ellos. Los programas van a ir terminando en distinto orden( no se puede asegurar en q orden van a quedar).

**Programa:** es una serie de pasos. Entidad pasiva en ejecución.

Si a lo anterior le sumamos, el valor del **contador de programa**, el valor de los **registros** y la **pila del proceso** (parámetros de funciones, direcciones de retorno, etc), obtenemos por definición un **Proceso**

(analogía pizzas)

**Ubicación del proceso en memoria:**



Stack → argumento para las **funciones**, direcciones de retorno. Por eso es dinámica.

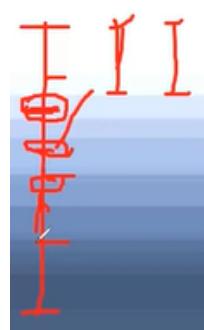
Heap → espacio para asignación dinámica de memoria (malloc())

Data → Variables globales, inicializadas

Text → código binario del programa

### Como se crean los Procesos/servicios?:

- En 2do plano o background → se inicializan con el sistema operativo, no se pueden ver ya q no tienen asociada los descriptores de ni la entrada ni la salida.
- Llamada a sistema **fork()**. Hace una copia de mi proceso. Se puede “dividir” el programa en varias partes, es decir mientras q se espera por una entrada o salida, se puede hacer una copia de otra parte del programa y seguir la ejecución:



- Cuando quiero hacer algo nuevo (“doble click”), o ejecutando un comando.
- Para sistemas de procesamiento por lotes.

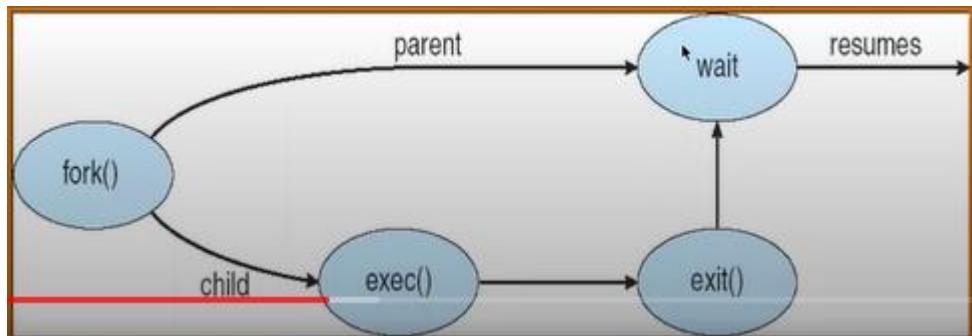
Ejemplo de creación de hijo con fork():

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int pid;
    printf("soy el padre\n");
    pid = fork();
    //hijo
    if (pid == 0){
        printf("soy el hijo\n");
        return 0;
    }
    sleep(200);
    printf("creo el hijo\n");
    return 0;
}
```

La ejecución de si se ejecuta primero el hijo o el padre lo determina el **planificador** del SO.

Alternativas de ejecución:



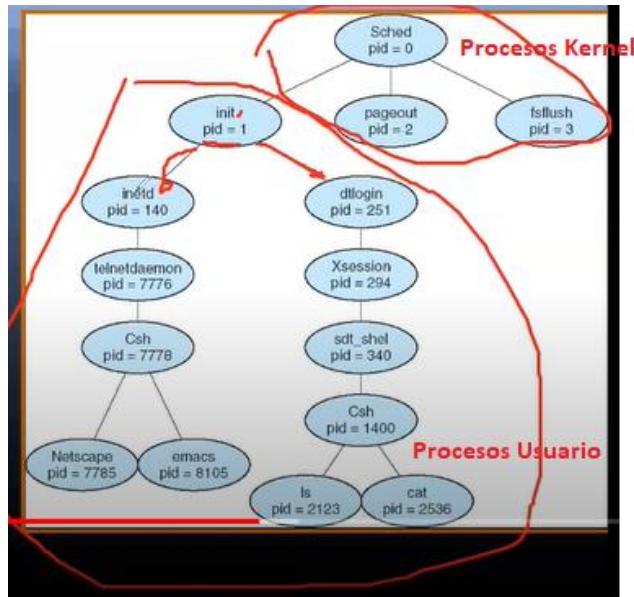
**Proceso padre:** puede esperar o no al proceso hijo. Lo espera con **wait(NULL)**

**Proceso hijo:** puede seguir ejecutando el mismo programa y datos q heredo del padre. O con **exec()**, le paso como parámetro un código binario para q ejecute eso y no lo q heredaría del padre. Para q ejecute otras instrucciones.

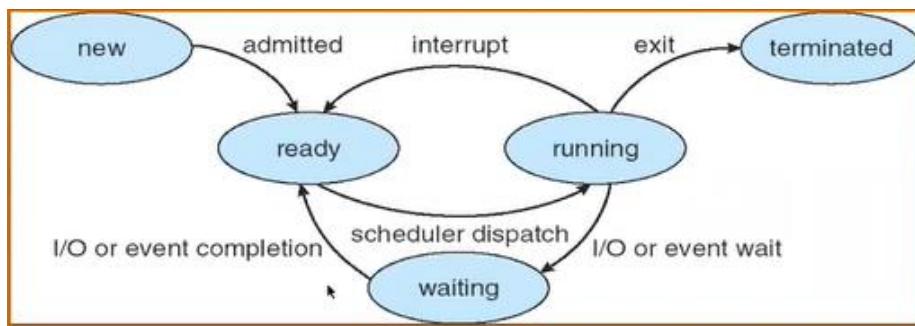
### Terminación de procesos:

- Terminación normal → tarea q realiza el SO, **exit()**,
  - si ingresamos el comando echo \$?, devuelve 0
- Terminación con error Voluntaria → un estándar error, devuelve distinto de 0
- Terminación con error Involuntaria →
  - No hay mas memoria
  - Violación de segmento
  - División por cero
- Recepción de una **señal** (1 evento por software):
  - Kill()
  - Handler de señales, con llamada a sistema signal()

## Jerarquía:



## Estados de los procesos:



Cuando un proceso es nuevo queda en una lista (**ready**). Cuando se estan ejecutando están en **running**, es decir que están usando la CPU. Si esta corriendo el proceso y hace entrada/salida, entra en modo **waiting**, es decir q se queda bloqueado. Cuando llega esa entrada o salida vuelve a estar en ready, hasta q vuelva a ser elegido por el SO, por una parte que se llama **Scheduler**.

Para q un proceso no haga mucho uso de la CPU cuando esta en running, se le asigna un tiempo con timers, y mediante una interrupción se lo vuelve a ready.

Cuando se esta ejecutando y hace un return 0, se termina.

(La transición entre running y waiting es una **llamada a sistema**)

(De waiting a ready es una **interrupción**)

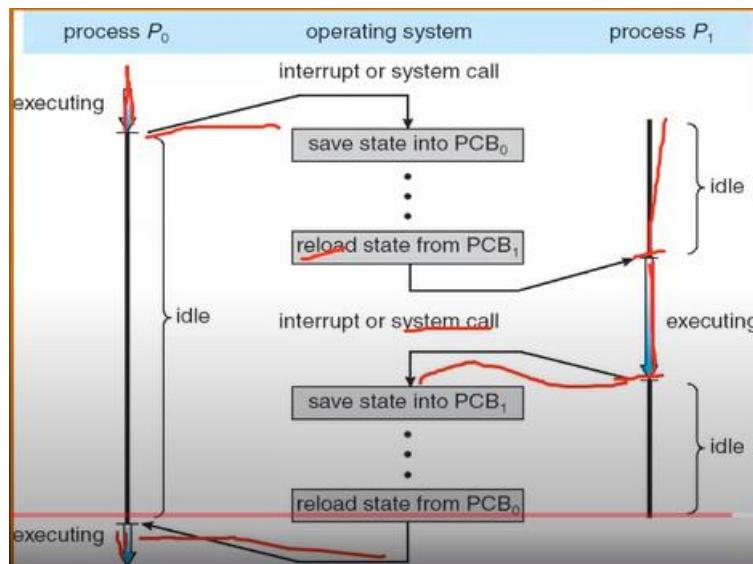
## PCB (Bloque de control de procesos):

¿Como el SO representa los procesos?

- PID (nº de proceso)
- TSS:
  - Estado
  - PC
  - Registros
  - Espacio de memoria q ocupa
  - Descriptores de archivos abiertos
  - Usuario/grupo
  - Etc



Cambio de contexto:



**Proceso zombie** → Proceso q se termino y se liberaron los recursos, solamente quedo su entrada en la tabla de procesos PCB.

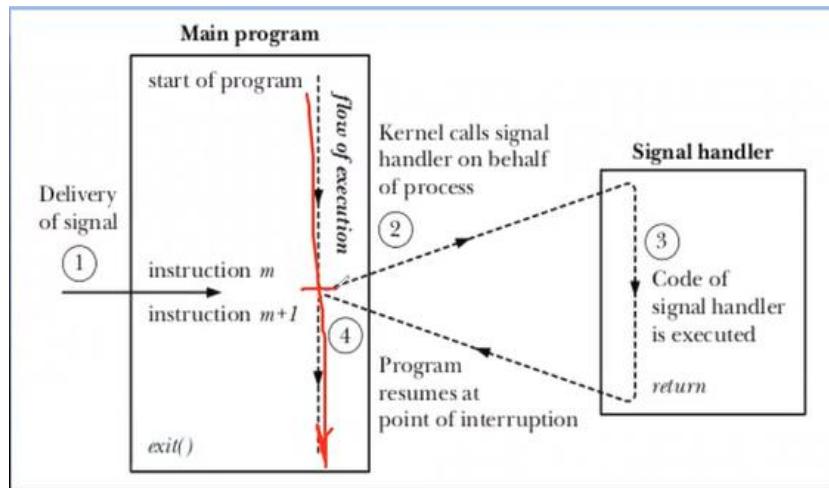
**Proceso huerto** → Cuando se termina el proceso padre, el proceso hijo se hereda al init o a un proceso superior.

## Señales:

Notificaciones asíncronas de software. Sirven para avisarle al SO de algún evento, avisar a un proceso de un evento por software o para controlar por teclado.

Cuando se recibe una señal, el proceso puede:

- Ignorarla
- Ejecutar una acción por defecto
- Ejecutar un manejador para la señal, programa encargado de “hacer algo” con la señal recibida:



Lista de señales:

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE       9) SIGKILL     10) SIGUSR1
 11) SIGSEGV    12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
 16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
 21) SIGTTIN    22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
 26) SIGVTALRM  27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
 31) SIGSYS     34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
 38) SIGRTMIN+4 39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
 43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
 58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
 63) SIGRTMAX-1  64) SIGRTMAX
```

Las señales **sigstop** y **sigkill** no se pueden ignorar.

## Clase 5: Inter-Process Communication

### Que es?

Una función básica de un SO. Permite enviar info entre procesos (datos o estado). Hay diferentes técnicas.

Necesito que los procesos se comuniquen para:

- Compartir info. (cuando 2 procesos q no son ni padre ni hijo ni nada quieren acceder a info de otro proceso)
- Acelerar los cálculos (+ en multicore)
- Modularidad (1 función x proceso)
- Conveniencia (1 usuario, varias tareas al mismo tiempo)

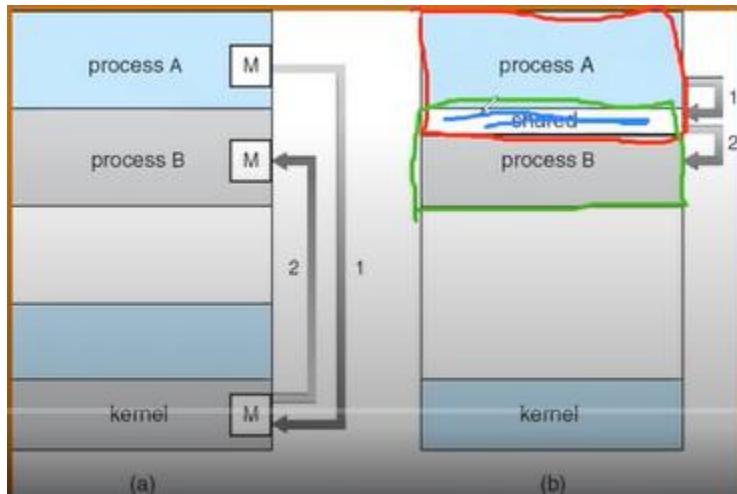
Ejemplo en consola, de conexión entre procesos:

```
ls -lR /home/carlost/historico/carlost/e-books/ | grep -i tane | cut -c 29-20
```

El carácter | (pipe) comunica la salida del primero proceso con la entrada del segundo proceso, y así sucesivamente.

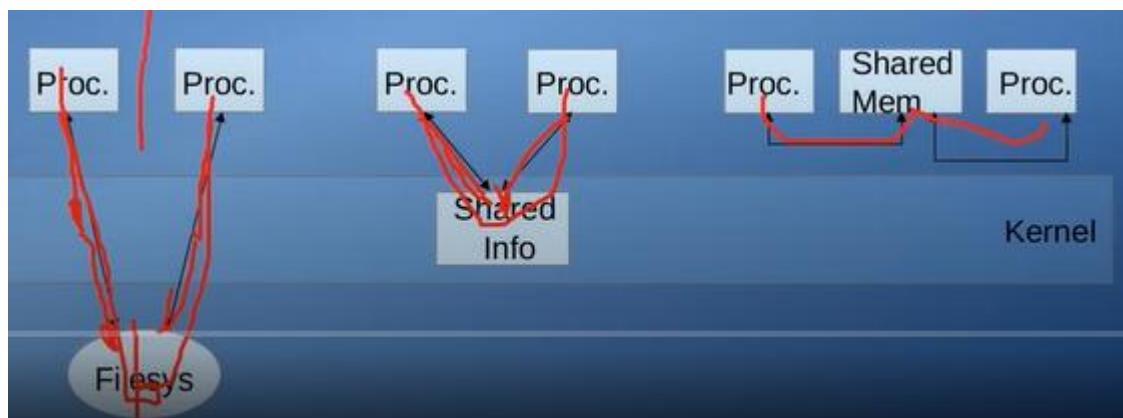
**Clasificación de IPC** (según como se comunican):

- Paso de mensajes:
  - Usando llamados al kernel.
  - Pequeñas cantidades de info
  - Fácil de implementar
- Memoria compartida:
  - Comparte espacio de mem
  - Rápida
  - Sincronizar



En memoria compartida, hay un espacio de memoria q es accedita por ambos procesos.  
Para hacer esto, los procesos deben hacer una llamada al SO para poder compartir un espacio de memoria.

- Paso de Mensajes:
  - Pipe (Tuberías anónimas)
  - FIFO (Tuberías con nombre)
  - Cola de mensajes
  - MPI (interfaz de paso de mensajes)
  - RPC (llamadas de procedimiento remotas)
  - Sockets
- Memoria compartida
  - Mapeo de archivos anónimos
  - Objetos de memoria
  - Archivos de memoria



1. Es un sistema de archivos: Archivo o Socket (en redes)
2. Kernel: Pipe, FIFO, Cola de mensajes

### 3. Memoria: memoria compartida. Este es el más rápido

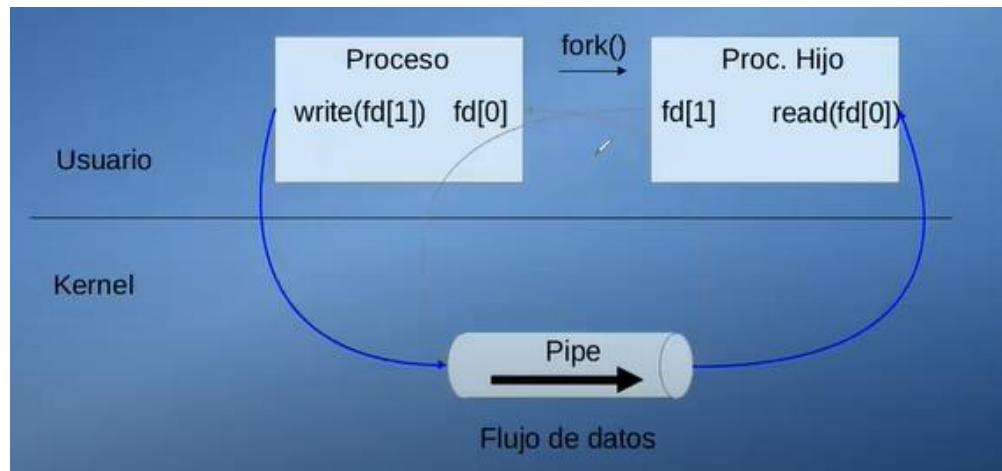
**Persistencia:** es el tiempo que permanece en existencia la información.

- Procesos: solo lo que dura este.
- Kernel: hasta cuando se reinicia la PC
- FileSystem: hasta q yo lo borre de memoria.

## PIPE

- Es un buffer en el kernel.
- Permite comunicar info entre procesos RELACIONADOS por herencia.
- Mensajes orientados a “stream” de datos (no segmentados) . Es decir orientados a flujos de bytes, por un lado escribo bytes y por el otro lado leo bytes.
- Es unidireccional.
- Se pasan dos enteros, dos descriptores. Uno para la entrada del pipe y otro para su salida. (teclado, pantalla, error)

Ejemplo de uso:



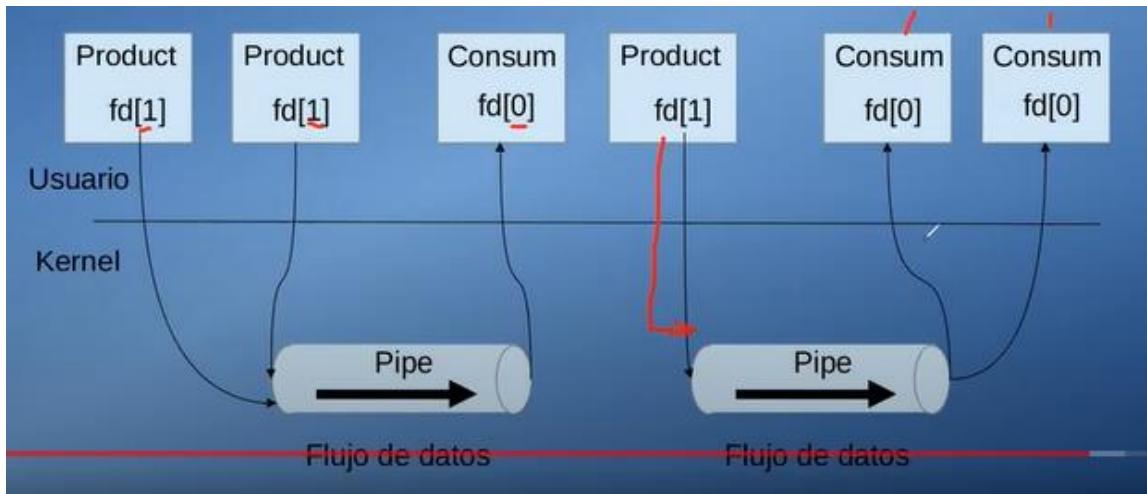
Primero se crea el pipe, luego se hace el fork()?

Si yo tuviera otro proceso, no tengo forma de acceder a esa tubería, porq no esta referenciada con un sistema de nombre ni nada. (X eso se llaman **Tuberías Anónimas**.)

## FIFO tubería Con nombre:

- Permite compartir info entre diferentes procesos no relacionados.
- Orientados a stream de datos, Flujo de bytes
- La tubería persiste con un nombre en el filesystem. Pero los datos no, es decir q si termina el proceso esos datos se pierden.
- mkfifo(nombre, permisos)

Problema de Stream de datos:



- No discrimina q productor escribió los datos consumidos.
- El primer consumidor lee los datos, y el resto no.

Estos problemas los tengo, porque esto no esta orientado a mensajes, o pasos de mensajes.

Algunas soluciones:

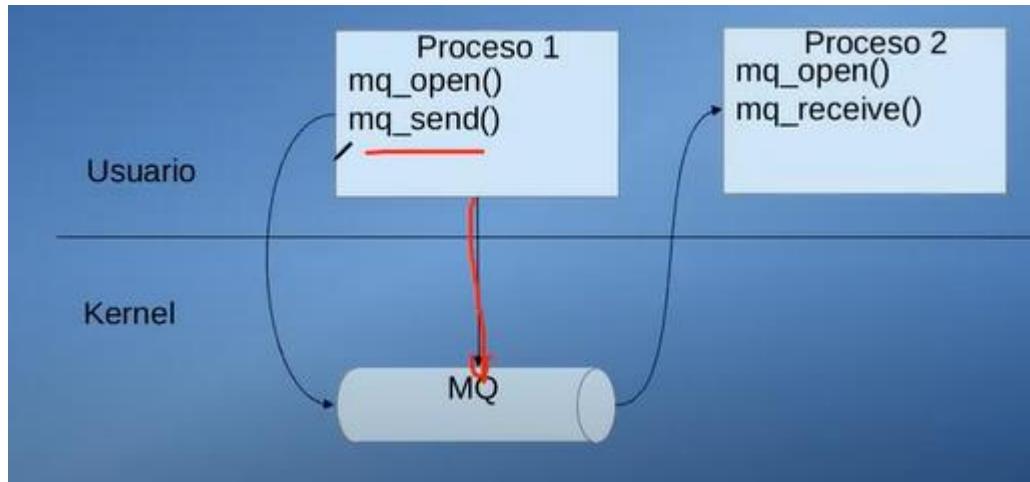
- Poner un encabezado.
- Un largo explicito (se envia primero el size)
- Largo fijo
- 1 registro por conexión. (para sockets)

## Cola de mensajes (posix):

- Para procesos también no relacionados.
- Como la persistencia de los datos esta en kernel (a diferencia de los anteriores), yo no necesito que se ejecuten a la vez los procesos.
- Tiene prioridades, si todos los mensajes tienen la misma prioridad, se leerán como FIFO, pero si no primero se lee el mensaje de prioridad mas alta, y así.

- Descriptor = mq\_open (nombre, banderas, permisos, ...)
- mq\_recieve/send(descriptor, info, longitud, prioridad)

Ejemplo de uso:



MQ es el descriptor que apunta la cola de mensajes.

## Clase 6: Sincronización

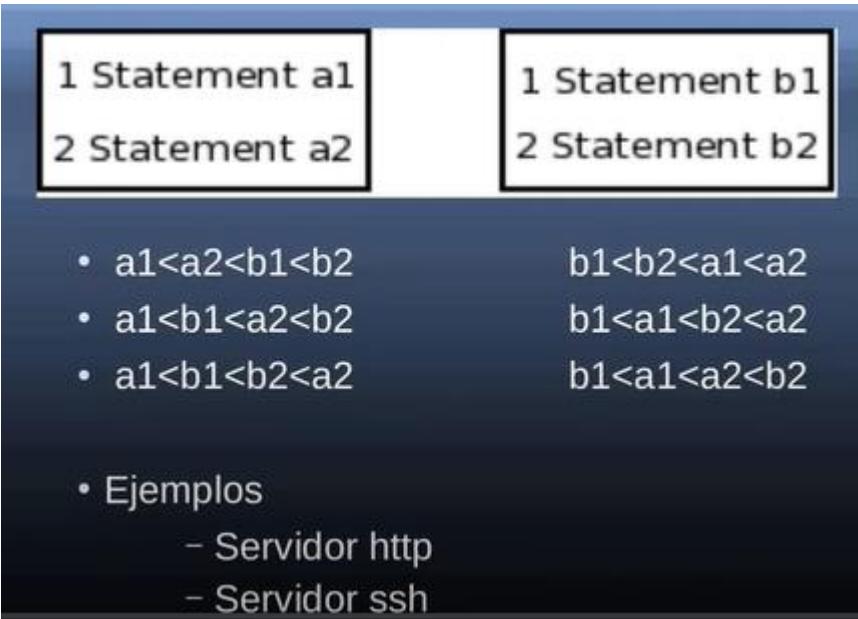
- Es la relación entre 2 o + eventos.
  - Antes
  - Despues
  - Durante
- Modelo de ejecución:
  - Sistema Monoprocesador Monotarea
  - Sistema Monoprocesador Multitarea (multiprocesos, multihilos)
  - Sistemas Multiprocesador Multitarea
- No determinístico:
  - No se ejecuta siempre igual (difícil depuración)
  - Programar PENSANDO en problemas de Sincronización.

Relaciones entre eventos:

**Concurrencia:** relación estado/evento. Donde es indistinto el orden en que se ejecuten los eventos, pueden ejecutarse a la vez.

- Se debe a multiprocesador o multiprogramación

- Generalmente NO modifican recursos compartidos
- El resultado SIEMPRE debe ser el mismo
- No se imponen restricciones de software.



( $<$  es menor en tiempo, es decir q se ejecuto antes.) Pueden ejecutarse en dichos órdenes.

Que no sea concurrente, quiere decir que por ejemplo en el servidor http, primero responde al cliente 1 y una vez que termina con este, ahí pasa al segundo, y así.

#### **Serialización:**

- 1 evento se debe producir antes q otro
- Generalmente 1 evento usa recursos modificados por otro
- Se deben imponer restricciones x software. Para q se ejecute primero el q necesito.
- Ejemplos:
  - Sistema de impresión
  - Cálculos intermedios
  - Depósito bancario

1 Statement a1  
2 Statement a2  
3 Statement a3

1 Statement b1  
2 Statement b2  
3 Statement b3

- Fijamos RESTRICCIÓN  $a_2 < b_2$
- $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$
- $a_1 < b_1 < a_2 < a_3 < b_2 < b_3$
- $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$
- $a_1 < b_1 < a_2 < b_2 < b_3 < a_3$
- El resto de las combinaciones inválido

#### Punto de encuentro:

- 1 evento debe esperar x otro evento e inversamente. Como una Doble Serialización. (ejemplo de ir a tomar algo con un amigo y nos juntamos en un lugar antes de ir, los 2 nos tenemos q esperar en dicho lugar para ir)
- Generalmente ambos eventos usan recursos modificados por otro
- Se deben imponer restricciones x software.

1 Statement a1  
2 Statement a2  
3 Statement a3

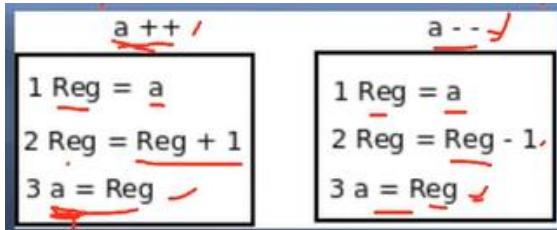
1 Statement b1  
2 Statement b2  
3 Statement b3

- Fijamos RESTRICCIÓN  $a_2 < b_3 \text{ AND } b_2 < a_3$
- $a_1 < a_2 < b_1 < b_2 < a_3 < b_3$
- $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$
- $a_1 < a_2 < b_1 < b_2 < b_3 < a_3$
- $a_1 < b_1 < a_2 < b_2 < b_3 < a_3$
- El resto de las combinaciones inválido

#### Exclusión Mutua:

- 2 eventos NO pueden ejecutarse a la vez (Ejemplo: 3 personas quieren entrar al baño, tonces tienen q entrar uno a la vez, no importa el orden.)
- Generalmente los eventos modifican recursos compartidos
- Resultado dependiente del orden de corrida
- Se deben imponer restricciones x software.
- Ejemplo: Escrituras concurrentes en base de datos

Ejemplo sin restricciones:



- SIN RESTRICCIONES ???
- a inicialmente es 5
- $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$  ( $a = \dots$ )
- $a_1 < b_1 < a_2 < b_2 < b_3 < a_3$  ( $a = \dots$ )
- $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$  ( $a = \dots$ )
- Que restricciones aplicaría ???

Por lo tanto, habría q poner de restricciones que se ejecute todo el a primero y después el b o viceversa.

Region Critica: es la zona q se protege cuando se esta modificando una variable para q otro proceso no pueda modificarla también.

Condición de competencia o carrera:

- Acceso a datos simultáneamente entre 2 o + procesos
- Resultado dependiente del orden de corrida
- Region/ Sección Critica
- Soluciones:
  - Software cerrojo
  - Hardware: desactivar interrupciones o asegura operaciones atómicas.

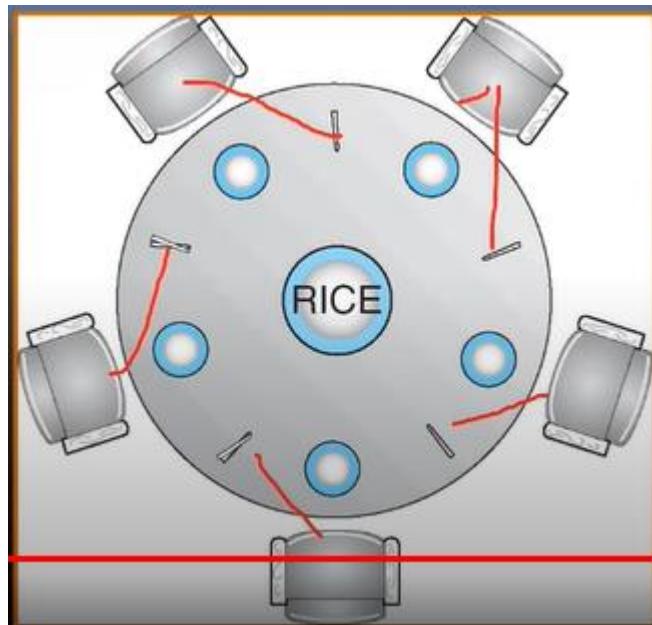
Por hardware:

- **TSL (Test and Set Lock)**
- **Semáforos**
  - Casi 1 varibale entera
  - Se puede inicializar en cualquier valor
  - Incrementar en 1 (post)
  - Decrementar en 1 (wait)
  - Primitivas son atomicas (usan TSL)
  - Decrementado, si resulta (-) el proceso/hilo se bloquea.
- **Mutex**
  - Casi 1 semaforo

- Solo puede tener como valor 1 o 0.
- **Barrier:**
  - Primitiva para punto de encuentro para N eventos. Hasta q todos no lleguen a tal punto, no pueden avanzar.
  - Se inicializa con N
  - Es atómica
  - Implementado con N mutex
- **Variable de Condición:**
  - Para evitar busy waiting
  - Se usa en conjunto con un mutex
  - Primitiva **wait** y **signal** (signal libera el mutex).

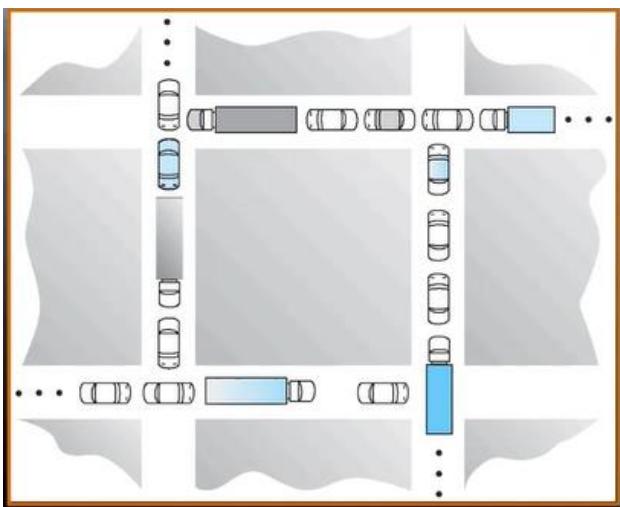
#### Problemas de sincronización:

- **Productor consumidor:**
- **Filósofos Comensales:**



(nadie termina comiendo en el ejemplo, o ejecutándose )

### Interbloqueo:



## Clase 7: Planificación

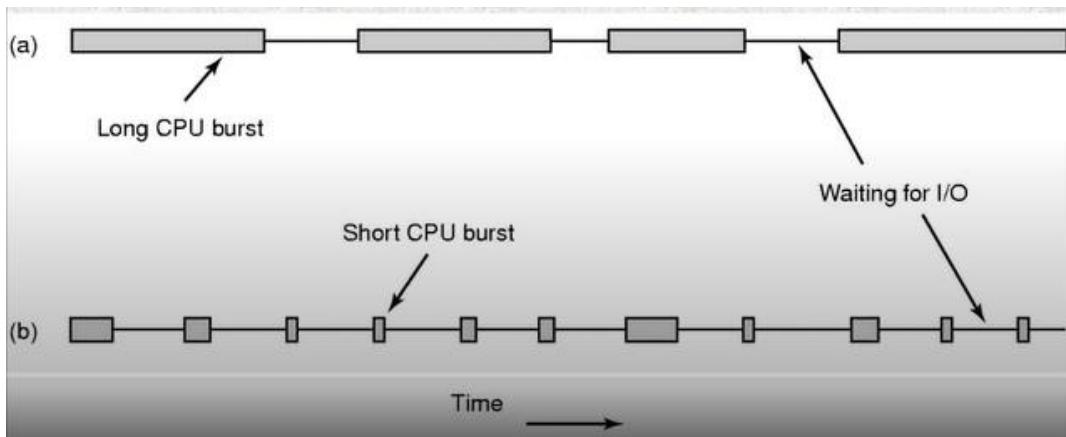
**Necesidad:** + de 1 proceso en ready

Cuando correr la planificación:

- Cuando se crea 1 proceso
- Cuando termina 1 proceso
- Luego q 1 proceso se bloquea (E/S gralmente)
- Cuando llega 1 interrupción de 1 periférico (Timer x ej)

**Tipo de procesos:**

CPU/Bonded (a) vs IO/Bonded(b)



## Tipos de algoritmos en función de interrupciones de clock:

- **No expropiativo:** Hasta q se bloquea o cede la CPU
- **Expropiativo:** Definen 1 maximo de tiempo de ejecución.

## Categoría de algoritmos:

- Batch
- Interactivos
- Real Time

### Sistema Batch

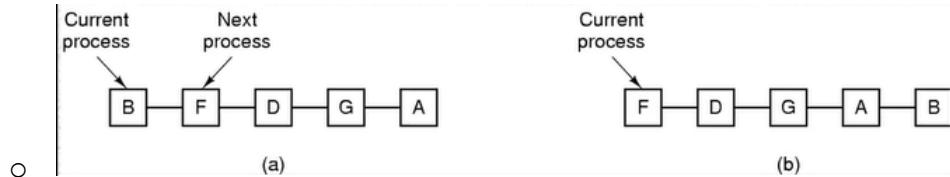
- **Fist-Come First-Served (X orden de llegada)**
  - Sistema no expropiativo
  - Se selecciona para correr el primer proceso encolado
  - Los siguientes se encolan el orden de llegada
  - Cuando el primero se bloquea, se selecciona para correr el segundo y se desencola.
  - Cuando se completa la I/O del primero, se encola nuevamente al final de la cola.
  - Facilidad de implementación
- **Shortest Job First** (el trabajo mas corto primero):
  - No expropiativo
  - Se conocen de antemano los **tiempos de corrida**
  - Se selecciona para correr le mas corto primero
  - Turnaround time(4 proc) =  $(4a + 3b + 2c + d)/4$
  - Disminuye el turnaround time si "a" es el mas corto
  - Ejemplo:



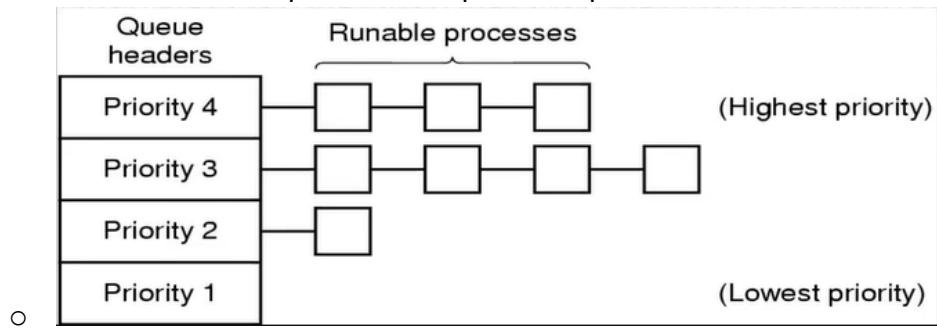
- **Shortest Remaining Time Next**
  - **Expropiativo**
  - Se conocen de antemano los tiempos de corrida
  - **Se selecciona el proceso al cual le queda menos tiempo para terminar**
  - Si llega 1 nuevo proceso se compara con el tiempo remanente del actual proceso corriendo.
  - Se puede suspender el proceso actual y correr el nuevo (X ser **Expropiativo**)
  - Los procesos cortos se ejecutan rápidamente.
  - Problemas → Inanición procesos largos, es decir q casi nunca se ejecutarían

## Sistemas Interactivos

- Planificación Round Robin:
  - Expropiativo
  - Se fija 1 intervalo de tiempo o quantum
  - A medida q llegan los procesos se encolan
  - Se selecciona el 1ero en la cola para correr
  - Si terminado el quantum sigue corriendo el proceso, se pone al final de la cola y se ejecuta el 2do
  - Facilidad de implementación.
- Problemas → Context switch, muchos procesos ingresan a la vez



- Planificación x Prioridad (con múltiples colas)
  - Expropiativo
  - Se fijan prioridades externas, dentro de cada prioridad, R. Robin
  - Se selecciona 1er proceso de + prioridad para correr
- Problema → Inanicion, debido a q las de menos prioridad pueden no ejecutarse casi nunca. (Por lo q se podría decrementar la prioridad a medida que corre)



- Shortest process next:
  - Mismo concepto q SJF. Cada comando es un Job
  - Problema para encontrar el proceso más corto a seleccionar
  - Aproximación basada en comportamiento pasado
  - Tiempo para un comando es T0
  - Siguiente corrida del mismo comando es T1
  - T promedio ponderado =  $aT_0 + (1-a)T_1$
  - $T_0, T_0/2+T_1/2, T_0/4+T_2/2, T_0/8+T_1/8+T_2/4+T_3/2$
  - Técnica de envejecimiento (fácilmente implementable, shift)

- **Planificación Garantizada**
  - **Expropiativo**
  - Si se tienen N procesos, se asigna a c/u  $1/N$  CPU
  - Se registra el tiempo q tiene derecho cada proceso(en teoría desde q inicio dividio N)
  - Se registra el tiempo q realmente a utilizado la CPU
  - Se almacena entonces el Ratio
  - **Ratio= Tiempo usado / Tiempo derecho**
  - **Se ejecuta 1ero el de menor ratio**
  - Problemas → Igual q el otro, tenes q estar guardando bastante info adicional
- **Planificación x Lotería:**
  - **Expropiativo**
  - Se dan números de lotería a cada proceso
  - De acuerdo a la importancia del proceso es la cantidad asignada, de manera de tener + probabilidad.
  - Si varios procesos cooperan, se pueden transferir los números entre ellos, aumentando la prioridad de alguno de ellos
  - Planificador selecciona 1 numero al azar
  - Facilidad de implementación.
- **Planificación x Parte equitativa:**
  - Es equitativo x usuario. Si 1 usuario tiene 1 proceso y otro tiene 10, no se repartirá la CPU de manera equitativa.
  - No interesa la cantidad de procesos q cada usuario tenga

## Sistemas en Tiempo Real

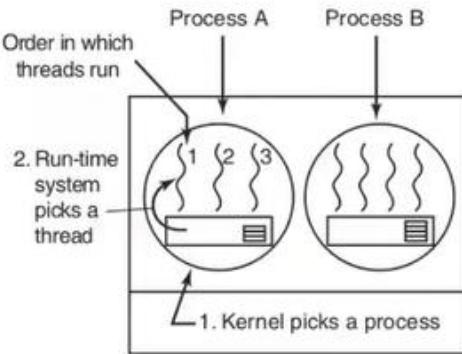
- **Planificación:**
  - Respuesta ante estímulos LIMITADA en el tiempo.
  - **Hard** real time: se deben cumplir siempre los deadtimes.
  - **Soft** real time: se hace lo posible x cumplirlos.
  - El planificador debe tener conocimiento exhaustivo del comportamiento de los procesos.
  - Dependiendo del tiempo de respuesta a los eventos es posible q no pueda manejarlos:  

$$\text{Si hay } m \text{ eventos periódicos y el evento } i \text{ tiene un periodo } P_i \text{ y requiere } C_i \text{ segundos de CPU , es planificable si:}$$

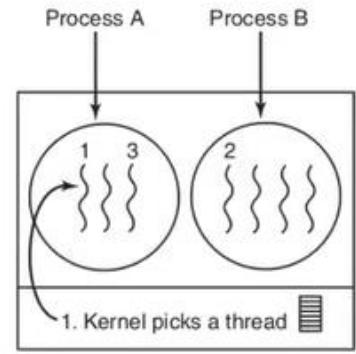
$$\sum_{i=1}^m (C_i / P_i) < 1$$

## Planificación de Hilos:

- Dependerá de la implementación de hilos
- La conmutación de hilos de usuario más rápida
- El kernel puede conmutar entre hilos de kernel del mismo proceso para no cambiar espacio de direcc.
- Los hilos de usuario pueden usar algoritmo customizado



(a)



(b)

- (a) Si el operativo usa algún algoritmo de planificación sabe q esta ejecutando el Proceso A y después tiene q pasar al B a lo sumo, pero no sabe q va a ejecutar el Hilo1 y después el 2 y así.
- (b) Con respecto a planificación en Hilos en Kernel, sabe en todo momento los hilos y la cantidad q hay en cada proceso.

Es + barato conmutar hilos del mismo proceso, q hilos de diferentes procesos. Ya q no necesita hacer **Cambio de Contexto** ni del Espacio de Direcciones ni del Espacio de Memoria.

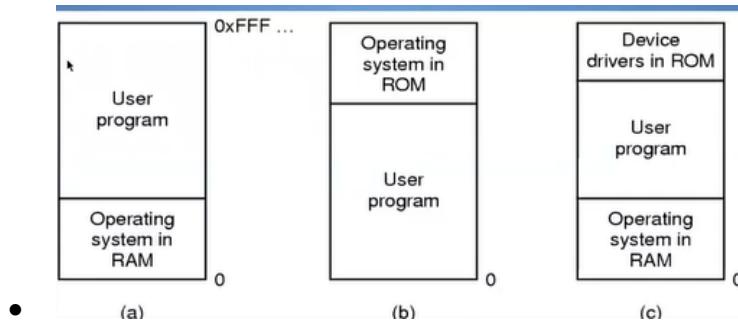
## Clase 8: Gestión de Memoria

### Administrador de Memoria

- Necesidad:
  - Se ejecutan programas SOLO en memoria principal
  - Se mejora la utilización de la CPU usando multiprogramación
  - Por lo tanto, se deben mantener VARIOS procesos en almacenados memoria principal simultáneamente
- Funciones:
  - Saber q partes están en uso y cuales no
  - Asignar + memoria a los procesos si las necesitan
  - Recuperar la memoria cuando los procesos terminan
  - Manejar el intercambio entre disco y memoria (cuando no entran todos los procesos en memoria)
- Esquemas
  - Monoprogramacion sin intercambio ni paginación
  - Multiprogramacion con particiones Fijas
  - Multiprogramacion con particiones variables-Intercambio

### Monoprogramacion

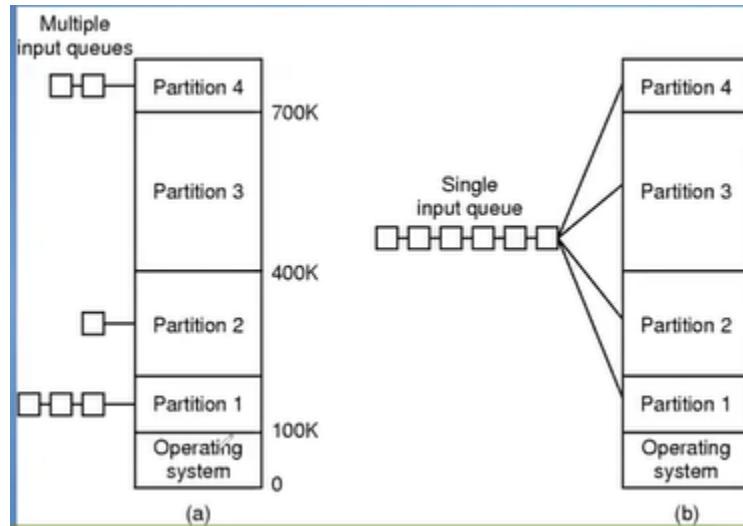
- Solo se ejecuta 1 programa a la vez
- Comparte la memoria el SO y el programa
- Cuando se escribe 1 comando:
  - SO carga el programa a la RAM
  - Le da el control al programa
  - Cuando termina da el control al SO



### Multiprogramación partes Fijas:

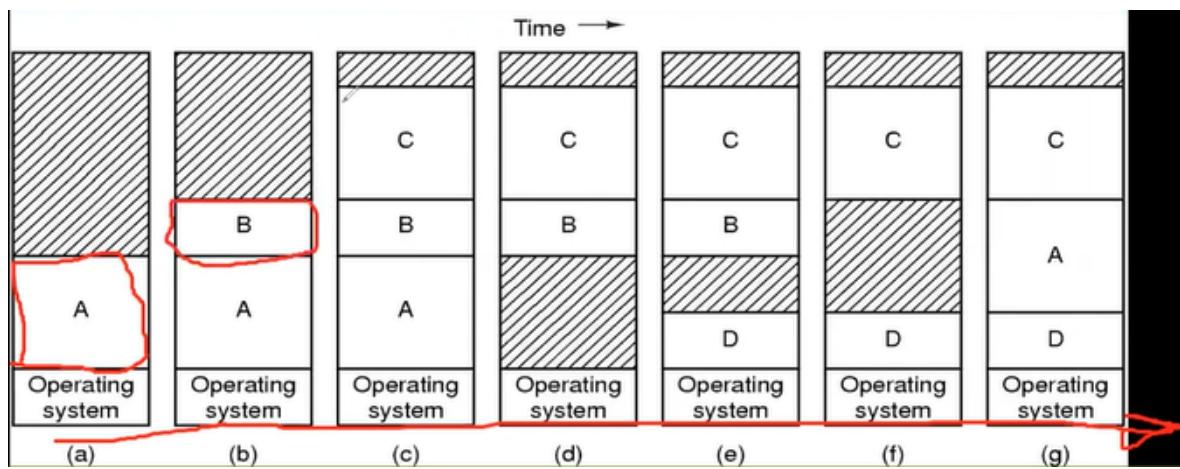
- Se divide la memoria en N particiones de distinto tamaño
- No varían en el tiempo. **Asignación estática.**
- Problemas:

- De reubicación de direcciones programa binario. (porque había q definirlos en una dirección dentro de la partición, para q si necesitábamos q accediera a un dato en tal dirección q sea dentro de la misma partición)
- De protección entre programas.



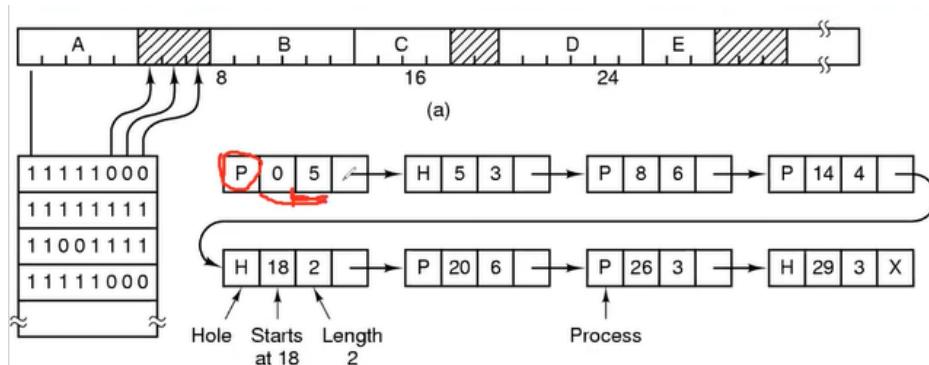
- Para asignar los procesos a las particiones, se utilizaban FIFOs, en el caso (b), se usaba una FIFO única, en el caso (a) se utilizaba una FIFOs por tamaños
  - **Múltiples colas (a):**
    - Procesos nuevos se encolan en partición  $\geq$  tamaño proceso
    - Desventaja:
      - Desperdicio dentro de las particiones
      - Particiones grandes sin uso, procesos chicos encolados
  - **Cola única (b):**
    - Algoritmos de elección del proceso a cargarse a memoria
      - Proceso + viejo q entra en partición
      - Examinar toda la cola, poner proceso + grande
  - Mejoras:
    - Utilizar registros de **Base** y **Límite**. (lo q veníamos viendo)
    - **Particiones Variables**

## Multiprogramación de particiones Variables. (intercambio)



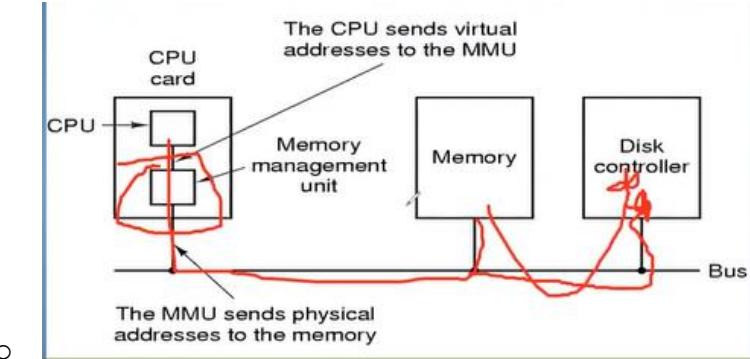
(En el eje horizontal es como va evolucionando en el tiempo.)

- Optimización de uso de la memoria → **Asignación dinámica**.
  - **Fragmentación externa** → Compactación (pero es un proceso lento). Se juntan los espacios chicos libres, para que quepa otro proceso. Se debe copiar ubicaciones de memoria en otros.
  - No hay reubicación de código de forma dinámica
  - El kernel DEBE tener registro de distribución de memoria
  - ¿Cuánto espacio de memoria asigno al programa?
  - **Fragmentación interna** (se asigna + memoria) Partes de la memoria dentro de la partición interna, no están en uso
  - ¿El procesador como sabe que partes de memoria internas no están en uso?
    - Mapa bits: cada bit representa una cantidad de KB de la memoria. (los 0 representan un espacio sin usar)
    - Lista enlazada: (en el ejemplo, primero hay un programa P, que empieza en la posición de la memoria 0 hasta 5, y apunta a un hueco H que empieza en 5 y termina en 3 +, y así).

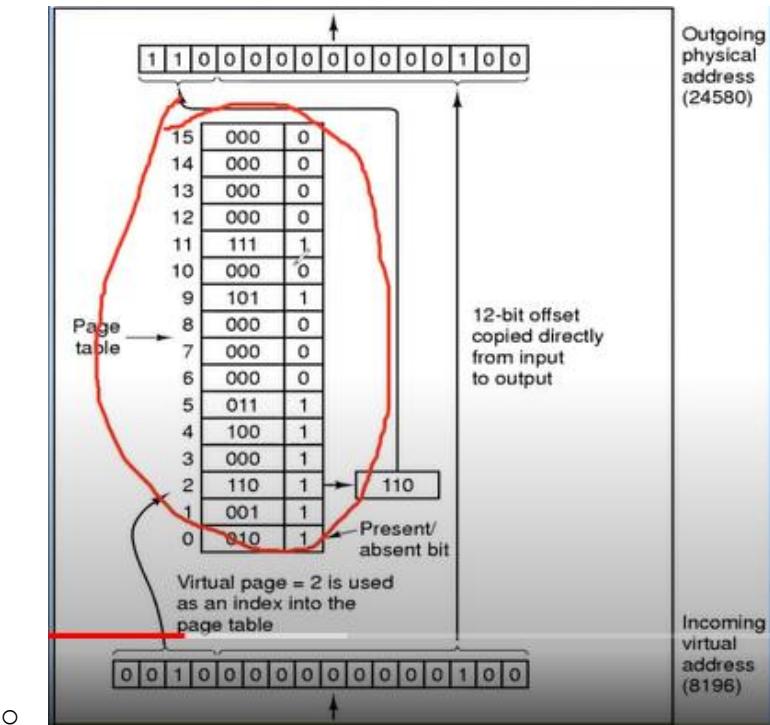


- En la lista enlazada, el proceso de encontrar un hueco es + rápido que en el mapa de bits. Ya que, en el mapa de bits, tiene q ir revisando bit a bit, para ver si hay ceros y si los hay tiene q seguir revisando bit a bit para ver cuanto espacio hay disponible. En cambio, en la lista, solo busca una H (q exista espacio), y ver el tamaño del mismo nada más.
- Pero en el mapa de bits, el proceso de actualizar los espacios libres, es + fácil y rápido. Porq solo hay q cambiar los 0 por 1.
- Estos ya no se utilizan + en la actualidad
- Algunos algoritmos para ubicación de memoria:
  - **First Fit:** 1er hueco  $\geq$  al espacio q necesita el proceso
  - **Next Fit:** Almacena ultima dirección ubicada
  - **Best Fit:**
    - Lento, recorre siempre todo el mapa o la lista
    - Genera mucha fragmentación externa pequeña
  - **Worst Fit:** simulaciones indican q no es muy buena idea
  - **Quick Fit:** distintas listas con tamaño estándar cada una de ellas.

## Memoria Virtual

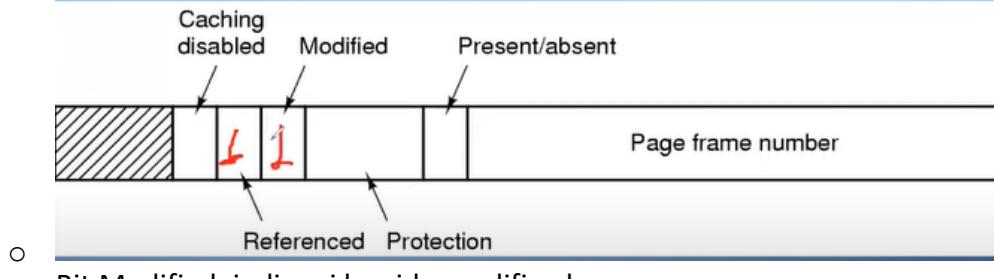
- Que es?
  - El concepto se refiere a que **el Espacio de Direcciones ya no es lo mismo que la Memoria Física**. Porque ahora ya no sabe el procesador con exactitud en donde realmente esta la info. Antes era una memoria soldada y era solamente eso.
  - The diagram illustrates the architecture of memory virtualization. It shows a CPU card containing a CPU and a Memory management unit (MMU). The CPU sends virtual addresses to the MMU. The MMU then sends physical addresses to the memory. The memory is connected to a bus, which also connects to a disk controller. Red lines and arrows highlight the flow of address translation from the CPU through the MMU to the physical memory.
  - El **MMU** es el encargado de trasladar o convertir las direcciones y direccionar la info en la memoria principal o en alguna secundaria.
- Implementaciones:
  - Paginación
  - Segmentación
  - Segmentación con paginación
- Técnica de Memoria Virtual:
  - Espacio de Direcciones → Páginas
  - Memoria Física → Marcos de Paginas

- Traducción entre ellas, MMU. La siguiente tabla la almacena la MMU:



- Detalles de Implementación genéricos:

- Campos de 1 tabla de páginas



- Bit Modified: indica si ha sido modificada
- Bit Referenced: indica si ha sido accedido
- Bit Present/Absent: indica si esta o no en un Marco de Pagina.
- Bit Caching disabled: indica q vaya siempre a leerlo

- Translation Lookaside Buffers (TLB)

- Tabla de paginas en memoria principal.
- Hardware q mapea paginas virtuales a memoria física (TLB)

Si quiero acceder a 1 espacio de memoria dentro de una pagina, y quiero ver en que Marco de pagina esta, y el bit q indica q esta presente o no, me dice que no esta, acá se produce un **Fallo De Pagina**.

Si estoy ejecutando un programa, y una parte de este está en otra página, distintas a las que están cargadas en la memoria, por lo q debo buscar otro marco de página, el problema es que **si están todos llenos**. Hay q buscar q página reemplazar:

### Algoritmos de Sustitución de páginas:

- **Not Recently Used Algorithm:** (el – usado recientemente)
  - En cada entrada de tabla de pagina se agregan 2 bits
  - Bit R (referenciado)
  - Bit M (Modified)
  - M es puesta a 0 cuando se carga el programa
  - R es periódicamente puesta a 0
  - Cuando se produce 1 fallo de página, se puede tener (se elige la página con menor clase):
 

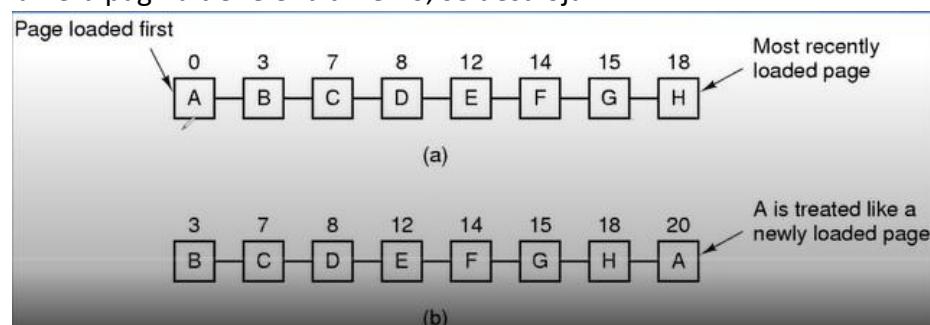
Clase 0	No Referenciada	No Modificada
Clase 1	No Referenciada	Modificada
Clase 2	Referenciada	No Modificada
Clase 3	Referenciada	Modificada

- **FIFO Algorithm:**

- Se implementa 1 Lista Enlazada:
- Cuando se produce 1 Fallo, se selecciona la 1era
- No se sabe si la pagina desalojada se usara a la brevedad.
- **No** es muy usado

- **Second Chance Algorithm:**

- Modificación a FIFO
- Se utiliza el bit R → Bit de accedido
- Si la 1era pagina tiene el bit R en 1, se pasa al final
- Si la 1era pagina tiene el bit R en 0, se desaloja



- **The Clock Algorithm:**

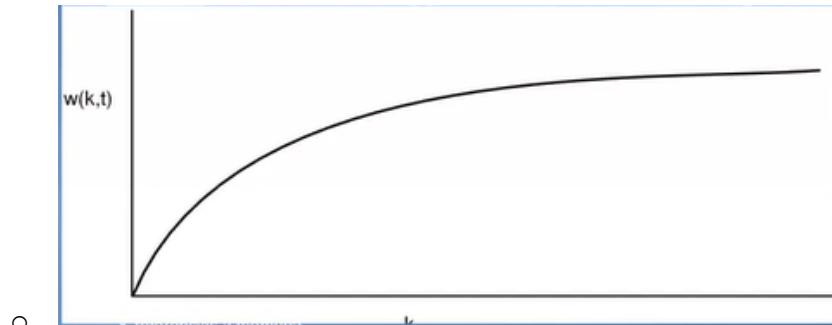
- Similar a FIFO o Second Chance
- No se mueven todas las paginas de la lista
- Si dejan las paginas en 1 lista circular

- La única diferencia con la anterior es el puntero q va apuntando los elementos de la lista. Así evitando copiar todos los elementos de la cola para estar trasladándolos.
- **The last Recently Used (LRU) Algorithm:**
  - Se busca la q + tiempo hace q no se accede (q no se usa).
  - Hardware especial. Matriz NxN, donde N es el numero de marcos de pagina.
  - A 1 toda la fila y luego a 0 toda la clomuna del marco usado

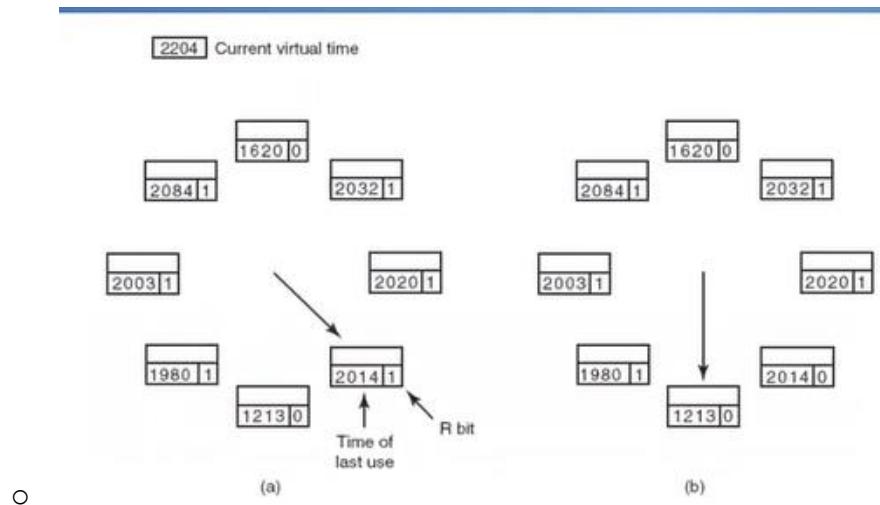
• Ej: 0 1 2 3 2 1 0 3 2 3

	Page 0	1	2	3		Page 0	1	2	3		Page 0	1	2	3		Page 0	1	2	3		Page 0	1	2	3
0	0	1	1	1		0	0	1	1		0	0	0	1		0	0	0	0		0	0	0	0
1	0	0	0	0		1	0	1	1		1	0	0	1		1	0	0	0		1	0	0	0
2	0	0	0	0		0	0	0	0		1	1	0	1		1	1	0	0		1	1	0	1
3	0	0	0	0		0	0	0	0		0	0	0	0		1	1	1	0		1	1	0	0
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)															

- **The Working Set:**
  - Principio de Localidad de referencia
  - Prepaging, carga de Working Set ANTES de correr
  - Implementación: Campo en Tabla Pagina Time of Last Use.



- **The WSClock Algorithm:**
  - Optimiza el algoritmo Working Set (no revisa todas las ent)



### Comparación de Algoritmos:

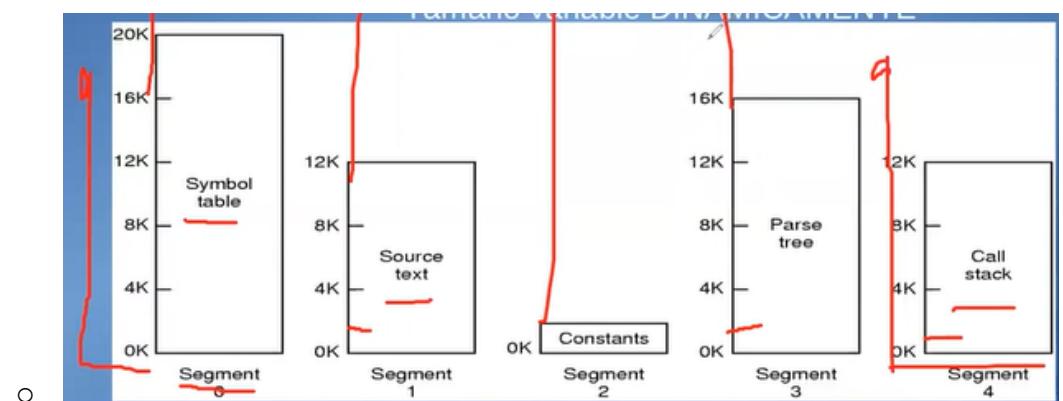
Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Otra técnica para resolver estos problemas:

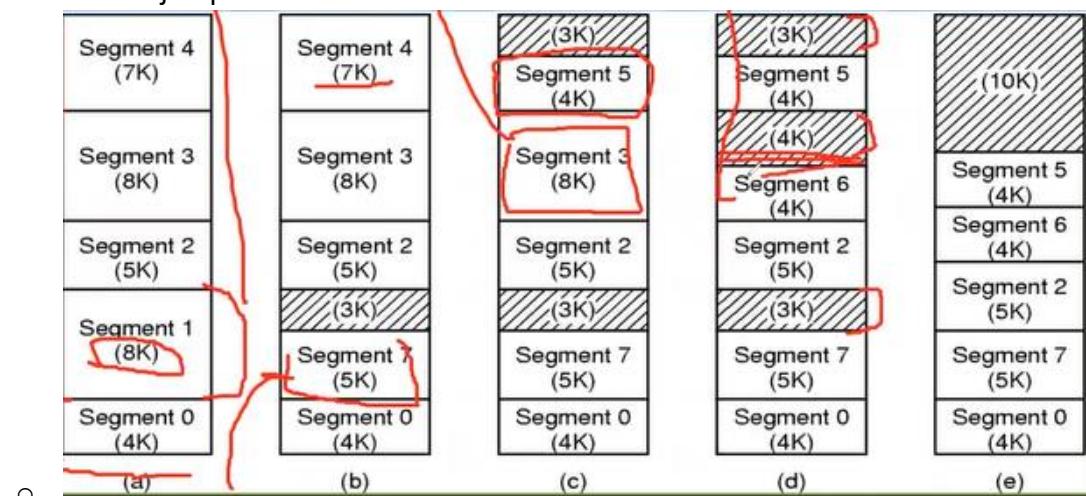
### Memoria Virtual Segmentación

El problema de paginación , es q hay 1 solo espacio de direcciones por proceso.

- **Segmentación:**
  - Le hago creer al CPU, q tengo N espacio de direcciones (Segmentos), independientes entre si y de tamaño variable DINAMICAMENTE:



- En un ejemplo:



- Lo que ocurre es q vuelven a aparecer **huecos**. Y además problemas de **Fragmentación Externa**.
- Esto se resuelve con **Segmentación Paginada**. Es decir , se hace paginación de la segmentación(se subdividen los segmentos en partes iguales), vamos a tener por ejemplo, el segmento 1 de 3 y así.

## Clase 9: Introducción a RTS

### Sistemas de tiempo real:

Aquel en el que el **resultado correcto** depende tanto de su  **validez lógica** como así también **del instante** en que se produce (determinismo)

**Determinismo:** El tiempo de ejecución de los programas de Tiempo Real debe estar **acotado** en el caso + desfavorable para cumplir las restricciones temporales. (No debe ser necesariamente rápido)

Velocidad/rapidez ≠ Determinismo

En sistemas de control puede haber hasta 4 niveles jerárquicos de software:

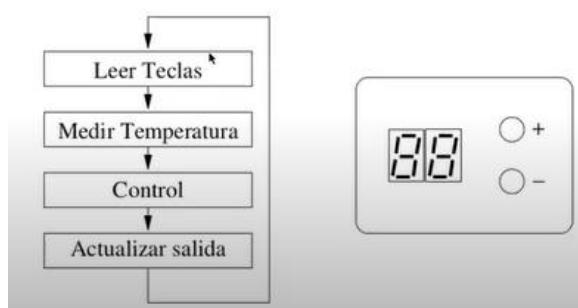
- Adquisición de datos / Actuadores: se ejecuta con 1 interrupción
  - Algoritmos de Control: se ejecuta periódicamente
  - Algoritmos de supervisión (trayectorias): periódicamente, pero con 1 periodo mayor q el anterior. Es una capa opcional de control a 1 nivel superior
  - Interfaz de usuario, registro de datos, etc:
- 
- 1 **Programa Secuencial** se divide en funciones q se ejecutan según 1 orden preestablecido
  - 1 **Sistema en Tiempo Real** se divide en tareas que se ejecutan en “**paralelo**” (sucesión rápida de actividades secuenciales → **Multitasking**)

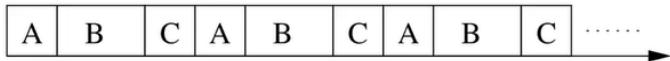
Técnicas para ejecutar tareas en paralelo:

- Procesamiento secuencial (scan loop): no es precisamente multitasking
- Interrupciones (Background/Foreground)
- Multitarea cooperativo (FreeRTOS)
- Multitarea expropiativo (FreeRTOS)

### Procesamiento secuencial:

Ejemplo en clase de termostato digital:





Tiene el problema de que si apretamos el pulsador 1 vez, al ser un bucle q se ejecuta muy rápido, es microcontrolador lo leerá muchas veces. Por lo q necesitamos de un tiempo de muestreo  $T_s$



Se puede utilizar un delay. El problema de agregar un  $T_s$ , q puede ser grande y no detectaría los pulsadores, es decir q hay **latencia**.

+ Problemas: Si el tiempo de ejecución de 1 tarea es > a  $T_s$ :



- **Pros:**

- Fácil implementación y depuración
- Fácil compartir datos, No hay q sincronizar

- **Contras:**

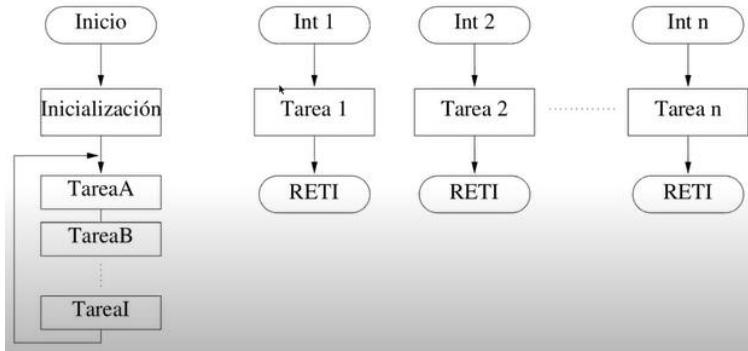
- Latencia asociada al bucle scan
- Difícil para tareas con distinto  $T_s$
- Solo para sistemas sencillos

### Interrupciones:

Esperar eventos externos Asincronos y disminuir Latencia.

2 tipos de tareas:

- Foreground
- Background (rutina de atención de interrupciones )



Ventajas:

Mucho menor la **latencia** en las tareas.

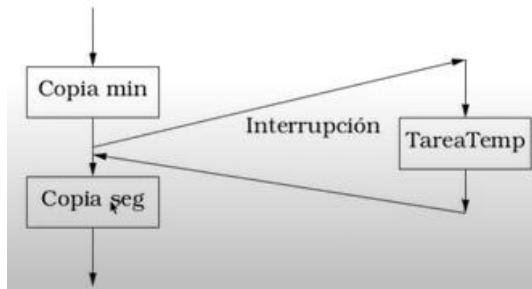
Se realizó el ejemplo en clase, de imprimir el tiempo (hora, minutos y segundos):

```

Hora: 00 : 01 : 58
Hora: 00 : 01 : 58
Hora: 00 : 01 : 59
Hora: 00 : 01 : 00
Hora: 00 : 02 : 00
Hora: 00 : 02 : 01 X

```

Pero, hubo **problema de concurrencia**. La tarea q imprime por pantalla seguía ejecutándose, mientras la interrupción todavía no actualizaba los minutos(se actualizaron los segundos pero no los minutos).



Solución a esto, es deshabilitar las interrupciones antes de hacer la copia de la hora anterior, dentro de la tarea en foreground de imprimir hora:

```

disable_irq(); // deshabilita interrupciones
copia hora = hora alt;
enable_irq(); // habilita interrupciones
sprintf (cadena, "%02d: %02d: %02d\n", copia hora.hor, copia hora.min, copia hora.seq);

```

Para q la copia se haga de manera **atómica**.

Si no, aplicar **RTOS**

## Clase 10: Intro FreeRTOS

Estados de las tareas:



Planificador:

En expropiativo, si 1 tarea tiene mayor jerarquía o nivel q otra tarea, puede sacarle uso del CPU.

- Planificador Cooperativo: tareas de 1er plano son las q llaman al planificador
  - En cada interacion de bucle
  - Cuando llevan demasiado tiempo ejecutando (yield)
  - Si no libero la CPU manualmente en ningún momento y se están ejecutando tareas con mucho procesamiento, pueden haber tareas q nunca se puedan ejecutar.
    - No hay problemas de concurrencia (coherencia), excepto si comparten datos 1er y 2do plano
    - + Sencilla la codificación
    - - Recursos
    - Difícil determinar la **Latencia** de las tareas 1er Plano.
- Planificador expropiativo:
  - 1 timer genera 1 **interrupción**, por lo q el planificador se ejecuta periódicamente.
  - Si antes de cumplirse el quantum, se bloquea la tarea, se ejecuta el planificador
  - Si antes de cumplirse el quantum hay 1 tarea de > **prioridad lista**, se intercambia su estado. (a diferencia de los SO de propósito general, q esperan)
  - El planificador elige la tarea lista de **Mayor Prioridad**.
    - Baja **Latencia** de las tareas 1er Plano
    - Hay q ser cuidadoso con la programación de las tareas
    - + Complicada la implementación

FreeRTOS:

- Software Con Licencia GPL
- Mayormente en C
- Puede trabajar de manera cooperativa o expropiativa
- Etc

### Tareas:

- Unidad básica de planificación. "Se implementan con funciones de C. Se debe devolver void y recibir 1 puntero a void como parámetro"
  - `void vTareaEjemplo (void *pvParametros)`
- NO DEBEN RETORNAR BAJO NINGUN CONCEPTO (no deben terminar)
- Si 1 tarea deja de ser necesaria, puede **eliminársela** explícitamente
- Si 1 tarea es necesaria, cualquier tarea puede **crearla** explícitamente

### Creación de Tareas:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,1
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

(puntero al nombre de la tarea, puntero carácter para debuggin, tamaño pila, puntero a void son los parámetros q recibe la función de tarea, prioridad, ID de la tarea)

- Se debe asignar a las tareas 1 tamaño de stack en WORDS:
  - `>= configMINIMAL_STACK_SIZE` (empieza en minúsculas porq es 1 Macro. Config porq dice en q archivo de configuración esta)
- Se pueden crear múltiples instancias de 1 misma función (con ≠ parametros)
- Prioridad: `tskIDLE_PRIORITY` a `configMAX_PRIORITIES - 1`. (IDLE función q se ejecuta con la mínima prioridad q no hace nada.)

### Ejecución de Tareas:

```
void vTaskStartScheduler( void );
```

### Planificador:

- **Fixed priority preemptive scheduling with time slicing:** no se va a cambiar las prioridades (**prioridad fija**), va a funcionar en modo **expropiativo**, y que tiene **quantum** de tiempo, por lo q puede cambiar de tarea.

`configUSEPREEMPTION = 1` y `configTIME_SLICING = 1`

(la segunda es: **configUSE\_TIME\_SLICING** )

- SIEMPRE ejecuta la tarea de > prioridad q esta en READY (no espera el quantum)
- Si esta en modo **time slicing**, en cierto tiempo el scheduler va a retirar la tarea en ejecución solo si hubiera 1 de = prioridad en READY (quantum)
- Se define 1 frecuencia llamada: **configTICK\_RATE\_HZ**. Es **inversa** al **quantum**
- Se puede **Suspender** y **Reanudar** las tareas.

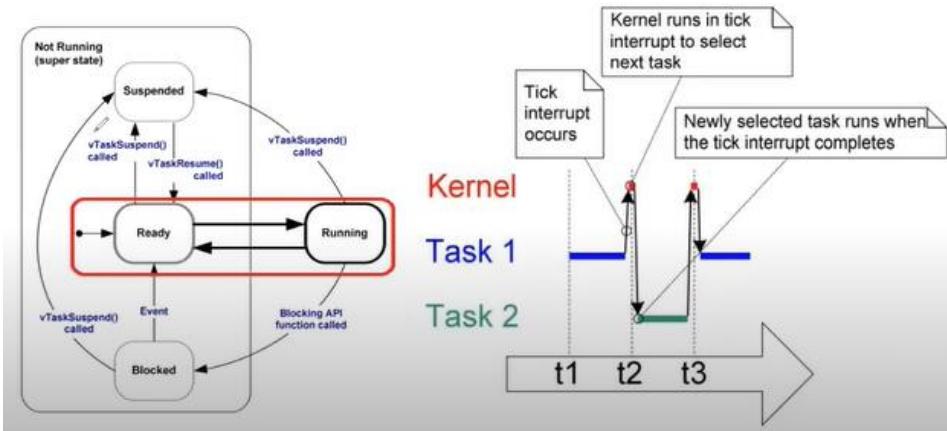
Ejemplo en clase:

Cada tarea prende y apaga el led, cada cierto tiempo, y manda una cadena de caracteres por puerto serie.

Hubo **problema de concurrencia**:

```
Tarea 2 is running
Tarea 1 is running
Tarea 2 is running
Tarea 1 is rTarea 2 is running
unning
Tarea 2 is runnTarea 1 is running
ing
Tarea 1 is running
Tarea 2 is running
```

En algún momento, se estaba mandando lo de una tarea y el planificador le cedió la CPU a la otra, por eso se ve cortado el mensaje en momentos. (Ya que ambas tareas tienen = **prioridad**)



**Tipos de tareas:**

- **Tareas Periódicas:** con frecuencia determinada. Son de ALTA PRIORIDAD, para tener baja latencia. La mayor parte de su tiempo están en estado Bloqueado. Se usa 1 timer y 1 contador.

- **Tareas Aperiódicas:** con frecuencia indeterminada. Tareas inactivas (bloqueadas), hasta q ocurre el evento de interés. De ALTA PRIORIDAD.
- **Tareas Continuas:** régimen permanente. BAJA PRIORIDAD.

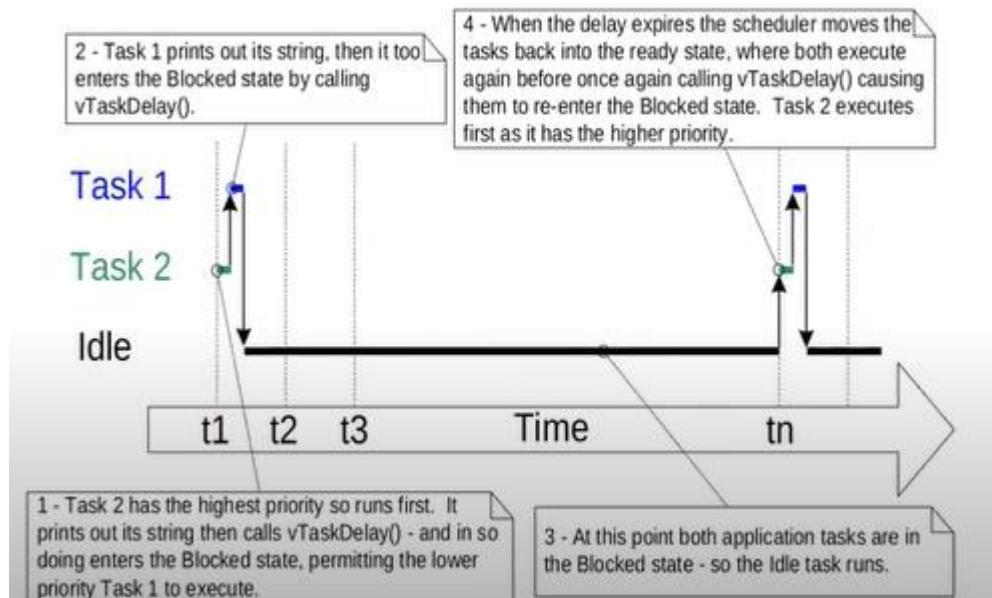
Recursos temporales para manejo de tareas:

`void vTaskDelay( TickType xTicksToDelay );`

Cede el control de la CPU mientras este tiempo no expira. Queda en estado BLOQUEADO.

Para expresar los milisegundos en TICKS: `pdMS_TO_TICKS()`

Se realizo un ejemplo en clase, donde tarea2 es de > prioridad q tarea1, ambas prenden 1 led. La tarea 2, después de cambiar el estado del led, ejecuta 1 TaskDelay, por lo q tarea1 puede ejecutarse:



Otra función para manejar tareas periódicas:

`void vTaskDelayUntil( TickType_t * pxPreviousWakeTime,  
TickType_t xTimeIncrement );`

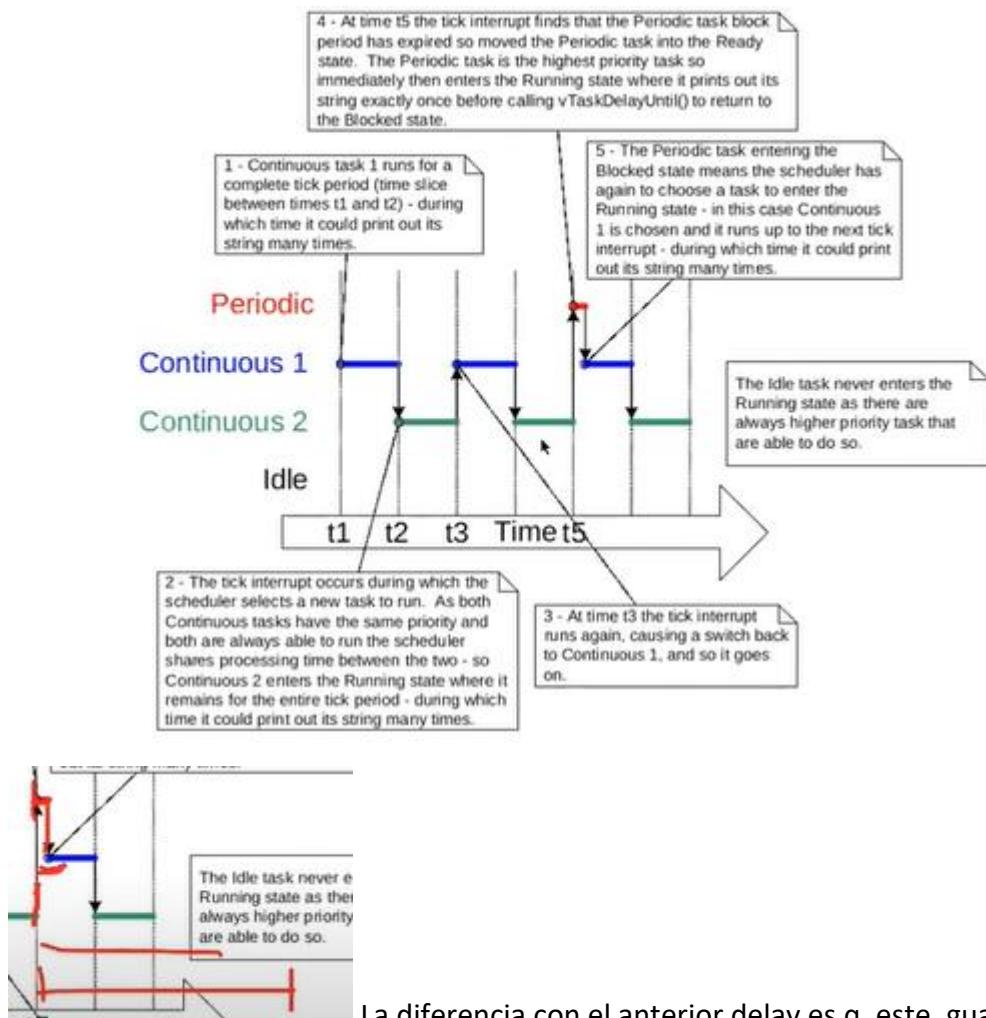
(el tiempo anterior de cuando llame a la tarea, y el tiempo de cuanto más quero q se quede dormida)

Asegura 1 tiempo constante entre **sucesos** llamados a esta función. También cede la CPU.

**Absoluto** → (`PreviousWT + TIincrement`)

#### Ejemplo 4:

Donde hay 2 tareas de = prioridad q se ejecutan alternadamente, y 1 tarea de > prioridad, q se ejecuta periódicamente.



La diferencia con el anterior delay es q, este, guarda desde q se ejecutó, y cuenta a partir de ahí, por lo q **siempre se mantiene el mismo periodo**.

Puedo **cambiar prioridades manualmente**:

```
void vTaskPrioritySet(TaskHandle_t pxTask,
                      UBaseType_t uxNewPriority);
```

Y con la siguiente podemos conocer su prioridad (pasandole su ID):

```
uBaseType_t uxTaskPriorityGet(TaskHandle_t pxTask);
```

## Clase 11: FreeRTOS

Recursos para Intercambio de datos:

Las tareas son funciones en C. **Reglas de visibilidad.** Se utilizan variables globales como intercambio (No hay IPC)

**Recursos de Sincronización de Tareas:**

- **Semáforos:**
  - Restringir el acceso a 1 sección particular del programa (**exclusión mutua**)
  - Ordenar cronológicamente eventos (**Serializar**)
  - Punto de encuentro
    - 2 primitivas: incrementar y decrementar
    - Se puede definir máximo tiempo de bloqueo
    - Semáforos binarios, mutex y counting
    - Se implementan con queues.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Se inicializa en **0**.

Para decrementar el semáforo:

```
BaseType_t xSemaphoreTake(  
    SemaphoreHandle_t xSemaphore,  
    TickType_t xTicksToWait );
```

Para incrementar el semáforo:

```
BaseType_t xSemaphoreGive(  
    SemaphoreHandle_t xSemaphore);
```

Debido a problemas de variables Globales. Se puede utilizar:

- **Cola de mensajes:**
  - Deben estar definidas globalmente
  - Se pueden bloquear al leer / escribir datos de 1 cola
  - Se debe definir tamaño de mensajes y cantidad de mensajes.
  - Pueden ser FIFO o LIFO.
  - Se puede definir máximo tiempo de bloqueo.

```
QueueHandle_t xQueueCreate(  
    UBaseType_t uxQueueLength,  
    UBaseType_t uxItemSize );
```

→ Para enviar mensajes:

```
 BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
    const void * pvItemToQueue,  
    TickType_t xTicksToWait );
```

Tiene TicksToWait para esperar un tiempo si se bloquea porq la cola esta llena.

→ Para recibir mensajes:

```
 BaseType_t xQueueReceive( QueueHandle_t xQueue,  
    void * const pvBuffer,  
    TickType_t xTicksToWait );
```

También tiene el tiempo por si se bloquea por esperar mensajes.

## Clase 12: FreeRTOS avanzado

- **Leer** datos de 1 cola:
  - xQueueReceive(): quita el dato de la cola
  - xQueuePeek(): lee pero NO quita el dato
- **Escribir** datos en 1 cola:
  - xQueueSend(): escritura normal a 1 cola
  - xQueueSendToBack(): escritura normal a 1 cola
  - xQueueSendToFront(): Pone el mensaje en la posición de salida. (apilar)
- **Consultar** los mensajes disponibles:
  - uxQueueMessagesWaiting()

**Espera x eventos (Colas o Semáforos):**

- **Espera Infinita:** se usa la macro: **portMAX\_DELAY**
- **Espera finita:** se usa 1 tiempo máximo de bloqueo:  
**tiempo\_ms/portTICK\_RATE\_MS** o **pdMS\_TO\_TICKS(tiempo\_ms)**
- **Sin espera:** usar 0 como blockTime

**IDLE:** FreeRTOS llama a la función Idle cuando esta ocioso. Este tipo de funciones tienen hooks, q permiten “enganchar” funcionalidad.

El hook q se llama cuando la App está Idle es → **void vApplicationIdleHook(void).**

Idle NO DEBE BLOQUEAR NI SUSPENDER, porq no quedaría ninguna tarea en estado Running, y siempre tiene q haber 1.

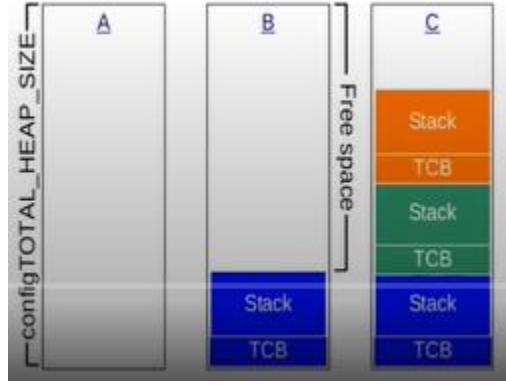
**Systick timer hook:** es útil para contar tiempo transcurrido sin dedicarle 1 tarea a esto.

```
void vApplicationTickHook (void);
```

Uso de Memoria Dinámica:

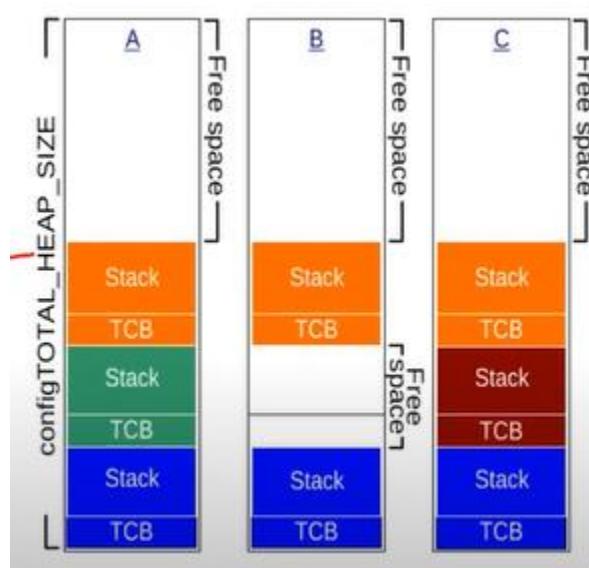
FreeRTOS asigna memoria dinámica para gestionar sus objetos cuando los crea:

- Tareas
  - Semáforos/mutex
  - Colas
- 
- Esta memoria la obtiene del heap definido en FreeRTOSConfig.h (configTOTAL\_HEAP\_SIZE), mediante llamadas similares a Malloc y Free.
  - El problema de Malloc, es q NO es determinístico, por lo q las aplicaciones en tiempo real no pueden permitirse el comportamiento del Malloc Estándar:
    - Tiempo de ejecución variable
    - Puede o no encontrar el bloque de memoria solicitado
    - Implementación muy abultada para la memoria disponible.
  - X Eso FreeRTOS separa la asignación de memoria del resto del kernel y la ubica en la capa portable. Podemos modificar el algoritmo de manejo de memoria en cada aplicación.
  - Proporciona 5 algoritmos de manejo de memoria:  
**Heap\_1.C, Heap\_2.C, Heap\_3.C, Heap\_4.C o Heap\_5.C**
- 
- **HEAP\_1.C :**
    - No contempla la liberación de memoria – ni implementa pvPortFree()
    - Se asigna secuencialmente la memoria hasta q se termina.
    - Es el + simple de usar. La memoria no se fragmenta.



- **HEAP\_2.C :**

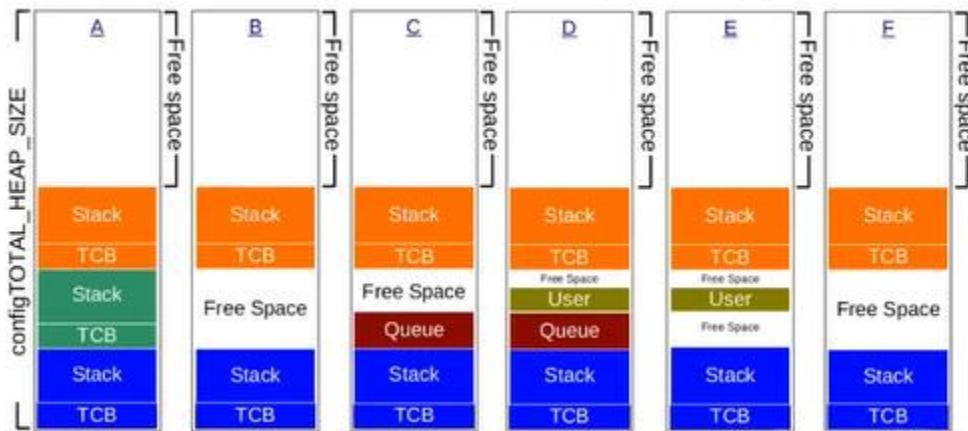
- Contempla la liberación de memoria
- Usa 1 algoritmo “best-fit”
- Recorre todos los bloques disponibles y devuelve el q + se ajusta a lo pedido. X eso **No es Determinista**
- EL sobrante del bloque asignado queda marcado como 1 nuevo bloque
- NO reúne huecos
- Es útil cuando se crean y destruyen distintas tareas pero CON EL MISMO TAMAÑO de stack.



- **HEAP\_3.C :**

- Usa la implementación estándar de Malloc y Free
- NO es determinista
- Encierra ambas operaciones en secciones criticas, haciéndolo thread-safe.

- No usa el tamaño de heap definido por config, si no el área de memoria definida para esto en el linkador.
- **HEAP\_4.C y HEAP\_5.C:**
  - La diferencia q tienen es q, el 4 es para 1 solo espacio de memoria y el 5 para n espacios (n memorias RAMs).
  - Contempla liberación de memoria, implementa pvPortFree().
  - Usa “first-fit” (el 1er lugar q encuentra)
  - NO es determinista
  - EL sobrante del bloque asignado queda marcado como 1 nuevo bloque.
  - **Combina bloques libres adyacentes**, disminuyendo fragmentación.



¿Cuál elegir?

- Si la totalidad de los objetos de la aplicación caben en la memoria disponible, **Heap\_1** es el + adecuado.
- Si se necesitan + tareas corriendo simultáneamente de las q se pueden albergar, se puede crear 1 tarea cuando se la necesita y eliminarla luego de su uso. **Heap\_4**
- Si se va a reciclar tareas con distinto tamaño de stack, se puede utilizar **Heap\_3**
- Si se tienen múltiples bancos de memoria, **Heap\_5**
- ¿Cómo los elijo? → **config.mk**

¿Cómo dimensionar el Heap?

[size\\_t xPortGetFreeHeapSize \(void\);](#)

- Llamarla luego de q se crearon todos los objetos del sistema
- Permite saber en cuanto disminuir su tamaño sin peligro.

Protección contra el desborde de pila:

El **high-water mark**, indica q tan alto llego el stack pointer en el stack.

```
UBaseType_t uxTaskGetStackHighWaterMark (xTaskHandle tarea);
```

Devuelve el mínimo disponible q hubo de stack desde q la tarea empezó a ejecutar.  
Además, se la puede usar para dimensionar el stack de la tarea.

### Chequeos en tiempo de ejecución:

FreeRTOS controla cuando la tarea sale de ejecución:

- Método 1: verifica q el stack pointer este dentro de la pila de la tarea al guardar el contexto (cuando sale de ejecución).
  - Es rápido
  - Si la pila desborda durante la ejecución pero vuelve 1 nivel permitido antes de salir de contexto, no lo detecta
- Método 2: verifica q los últimos 20 bytes de la pila mantengan su valor inicial.
  - + lento pero + seguro
  - Solo fallaría en el caso de q el desborde de la pila tenga el mismo patron q el valor inicial.

Con [configCHECK\\_FOR\\_STACK\\_OVERFLOW](#), se elige 1 o 2.

### Control de estado de las tareas:

Se generan estadísticas de tiempo de ejecución de las tareas.

- Ajuste y depuración SOLO durante la fase de desarrollo
- vTaskGetRunTimeStats()
- UxTaskGetSystemState()

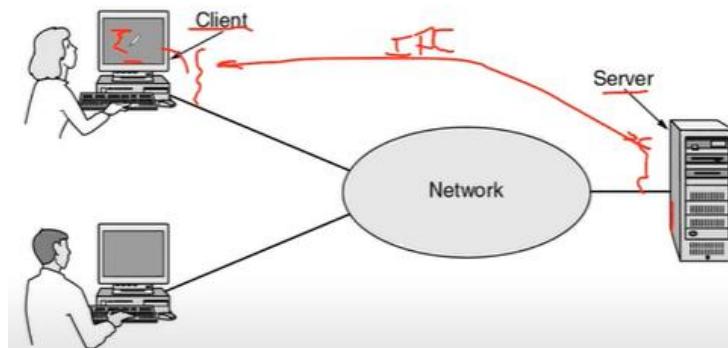
## Clase 13: Intro Redes de Computadoras

- Las redes de computadoras son Autónomas interconectadas
- Utilizan una misma tecnología de conexión
- Internet → Son muchas REDES de computadoras interconectadas. (NO 1 sola red)
- **Sistemas distribuidos**, puede estar distribuidos en 1 o + redes (red o red de redes).  
Cuando nos conectamos a internet, nos conectamos a 1 Software q utiliza 1 protocolo llamado “World Wide Web” (www).

- Se utilizan para:
  - Business Applications
  - Home Applications
  - Mobile Users

### **Business Apps:**

Cliente y servidor son **procesos**.



(utilizan Ipc)

### **Home Network APP:**

- Peer to Peer (torrent) (no hay rol fijo de quien servidor o cliente)
- Person to Person (Email, Chat, VoIP, Social Networks)

### **Mobile Network Users:**

Wireless	Mobile	Applications
No	No	Desktop computers in offices
No	Yes	A notebook computer used in a hotel room
Yes	No	Networks in older, unwired buildings
Yes	Yes	Portable office; PDA for store inventory

### **Tipos de tecnología:**

1. **De difusión (Broadcast Links)**
2. **De punto a punto**

Generalmente las de difusión se utilizan localmente y las de punto a punto como larga distancia.

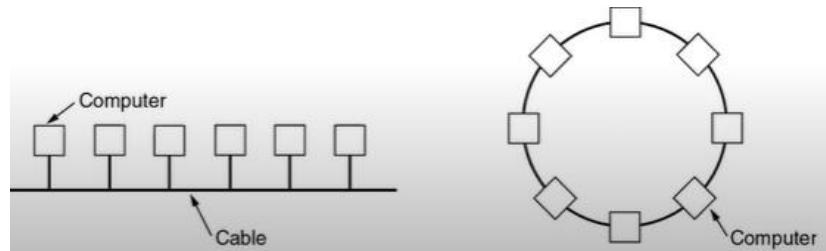
### **Tipos de áreas:**

- **Local Area Networks**
- **Metropolitan Area Networks**

- **Wide Area**
- **Wireless Networks**
- **Home Networks**
- **Internetworks**

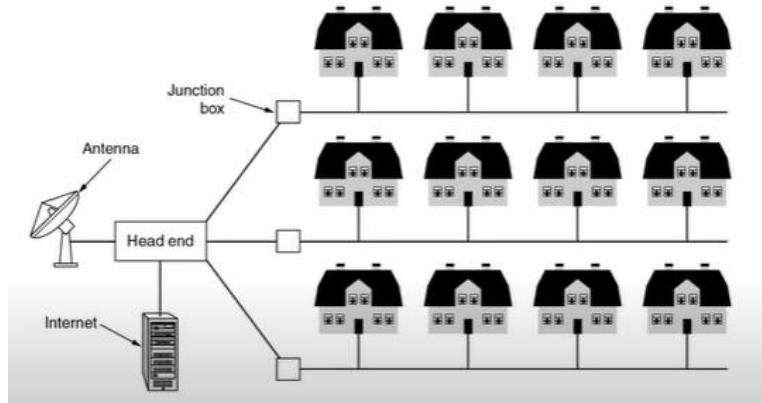
### **Local Area Networks:**

- Diseño estático (prácticamente no se usa)
- Diseño Dinámico
  - Todas interconectadas
  - O forman 1 anillo

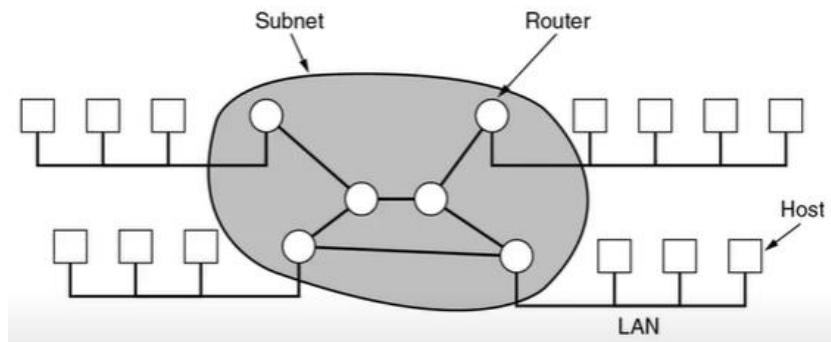


### **Metropolitan Area:**

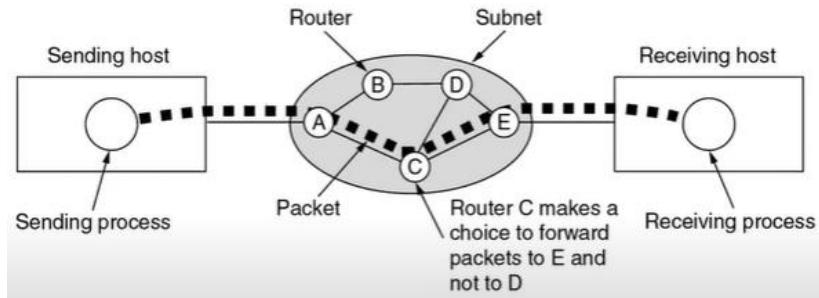
X ejemplo 1 red de TV x cable:



### **Wide Area (red de área amplia):**



Conexión de distintas redes lejanas, mediante una sub red.



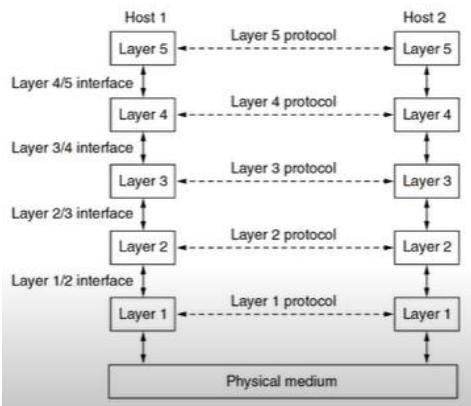
La info va en **paquetes**.

### Wireless Networks:

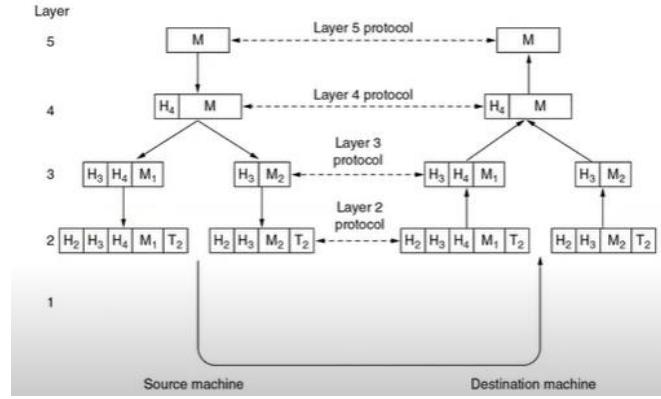
- System interconnection
- Wireless LANs
- Wireless WANs

## Network Software

### Jerarquías de protocolos:



Ejemplo + genérico:



Donde T (tail), H(head). En la capa 3 en este ejemplo se subdivide la info en 2 porq no puede mandar toda la info junta(tamaño).

### Problemas q deben tratar las capas:

- Esquema de nombre de direccionamiento (en todas las capas)
- Control de error
- ¿Hago control de flujo? (Si 1 maquina es + rápida q otra)
- ¿Q pasa si el canal es compartido? (multiplexo o no)
- Routing. Si tengo q reencaminar la red, por q camino lo hago
- Calidad de servicio.
- Seguridad. Confidencialidad, autenticación, integridad.

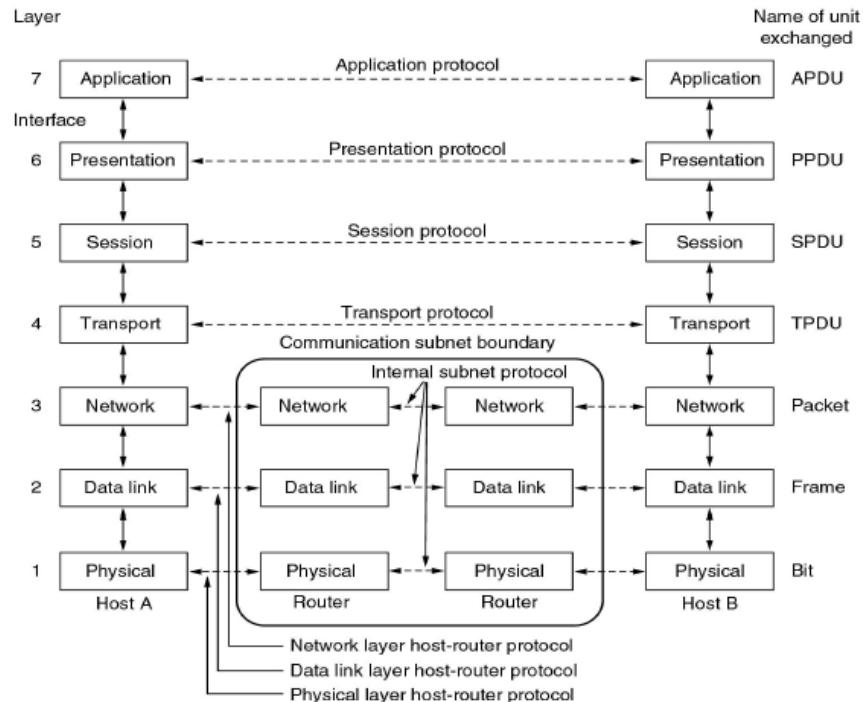
### Servicios y protocolos:

- **Protocolo** → estándar q tienen q seguir las mismas capas, para prestar 1 servicio.
- **Servicio** → servicio o información q ofrece o le agrega a la capa superior

### Modelo de referencias:

- **OSI:**
  - Modelo de 7 capas
  - En la actualidad no se utiliza
- **TCP/IP**

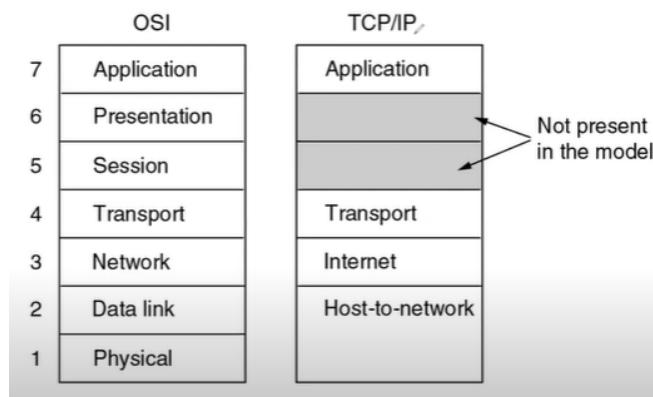
### Modelo OSI:



1. Capa física
2. Capa de enlace (comunicación de 1 misma red)
3. Capa de red (Esta capa opera cuando hay q trabajar en 2 máquinas q no están en la misma red.)
4. Capa de transporte (cual de todos los procesos q tiene corriendo es la q tengo q mandar información.)
5. De sesión (asegurar ser quien digo ser. Iniciar 1 sesión antes de enviar datos)
6. De presentación (como se presenta la info. Si es ASCII o q)
7. Y aplicación (es el proceso q se está ejecutando)

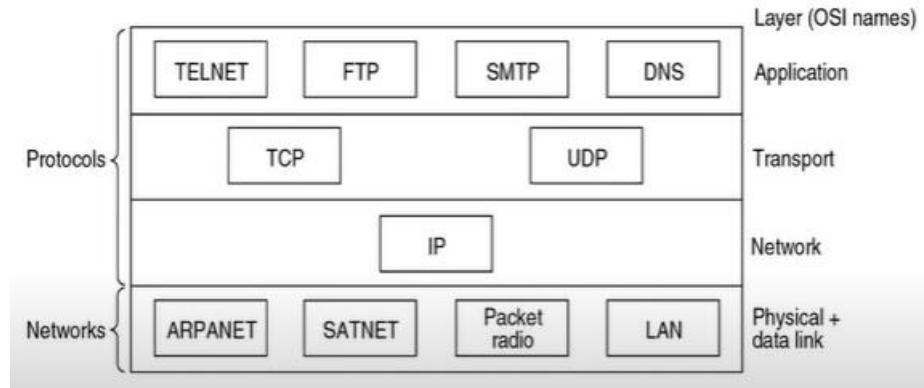
El **router** es el intermediario.

#### Diferencia entre OSI y TCP/IP:



Lo de presentación lo resuelve la aplicación y la parte de sesión lo resuelve la capa de transporte.

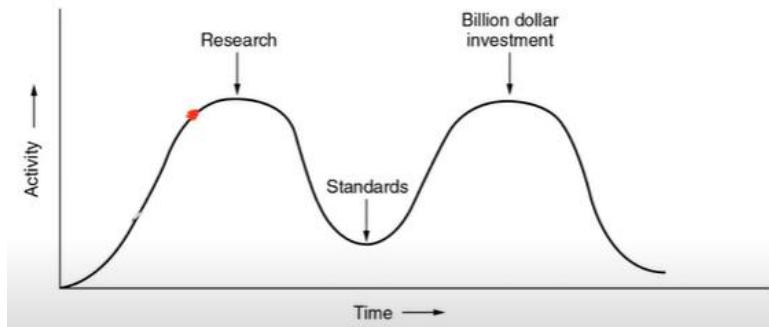
Además, se podría utilizar cualquier Capa Física y Capa de Enlace.



(capa de enlace y física podrían ser cualquier protocolo en realidad, esos son ejemplos)

¿Porq no OSI?:

- Mal timing (sacaron el estándar antes q salieran los protocolos q cumplían con esto)



- Mala tecnología
- Mala implementaciones
- Malas políticas

Críticas al TCP/IP:

- No separa claramente entre servicio, interfaz y protocolo.
- No es concretamente 1 modelo.
- Al no estar separados en capas realmente, es + difícil reemplazarlas, como ocurría en OSI. (fue todo 1 bardo reemplazar la capa de IP por la IPv6)

## Clase 14: Capas de Enlace

En la **capa física** viajan bits o símbolos, y además proveer 1 interfaz física en la red.

La interfaz estándar define:

- La codificación
- Que técnica de transmisión
- Que medios (medio de transmisión, medio de conexión)

### Capa de Enlace:

Tiene como **finalidad** → Transmitir entre máquinas de la misma red. (con el mismo canal de comunicación, es decir lo mismo q sale entra)

#### Limitaciones:

- Errores
- Diferencia en bit rates (cuando x ejemplo el emisor es + rápido q el receptor)
- Delay

#### Servicios q otorga:

- Entramado (**Framming**)
- Manejo de error
- Y manejo de flujo

Virtualmente la capa de red ve q la comunicación la hace la capa de enlace, pero realmente, la de enlace le agrega info para q la capa física transmita la info.

#### Tipos de servicios:

- **Sin confirmación y sin conexión**
  - Para sistemas con baja tasa de errores
  - No hace falta corregir los errores
  - 802.3 - Ethernet
- **Con acuse de recibo (con confirmación) y sin conexión**
  - Se manda el dato directamente, sin establecer 1 conexión previa
  - Canales no tan confiables
  - Si hay error, el emisor retransmite (ineficiente)

- 802.11 - Wifi
- **Con acuse de recibo y 1 comunicación orientada a conexión**
  - Para evitar la ineficiencia anterior, cada mensaje q transmito lo enumero, por lo q la reconfirmación la hace con ese numero. Por lo q se hace la retransmisión de 1
  - En canales largos (satélite y telefono), se utilizan estos
  - 3 fases

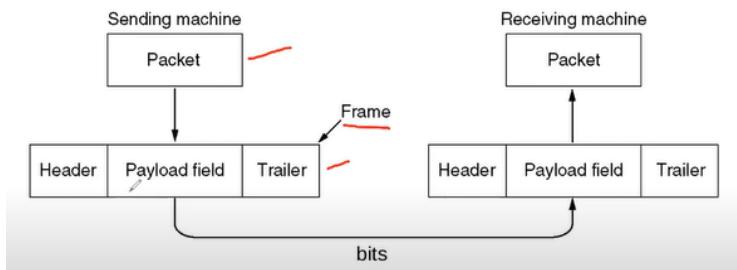
### Framming:

Tomar el mensaje y partirlo en tramas. Esto se hace porq no se sabe el tamaño del mensaje, por lo q se debe partir para agregarle la información de la capa de enlace al mensaje, sobre control y corrección de errores.

Indicar inicio de trama:

- Conteo de Bytes
- Byte bandera
- Bit bandera
- Y violación a la capa física.

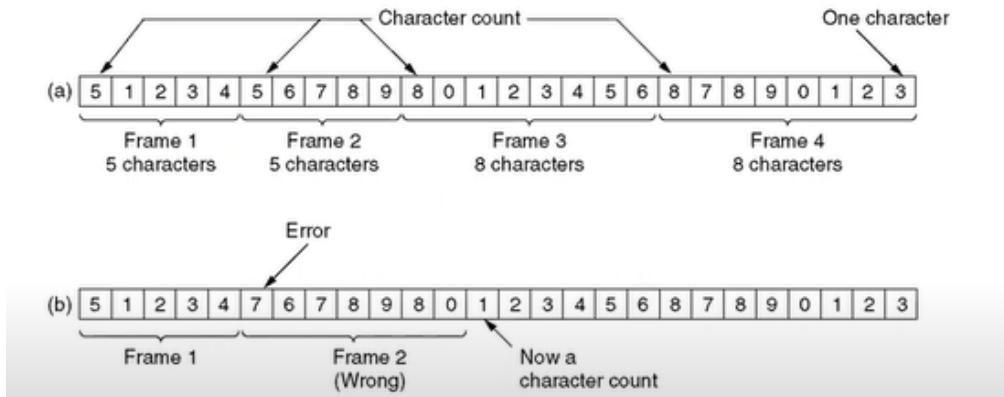
(En capa de enlace, el mensaje es **Trama o Frame**, y en la capa de red, el mensaje es **1 Paquete.**)



- **Bytes Counting** → Es muy malo, no se usa, ya q 1 error en el encabezado, donde se especifica el tamaño de la trama, te desincroniza todo.

(a): sin error

(b): con error

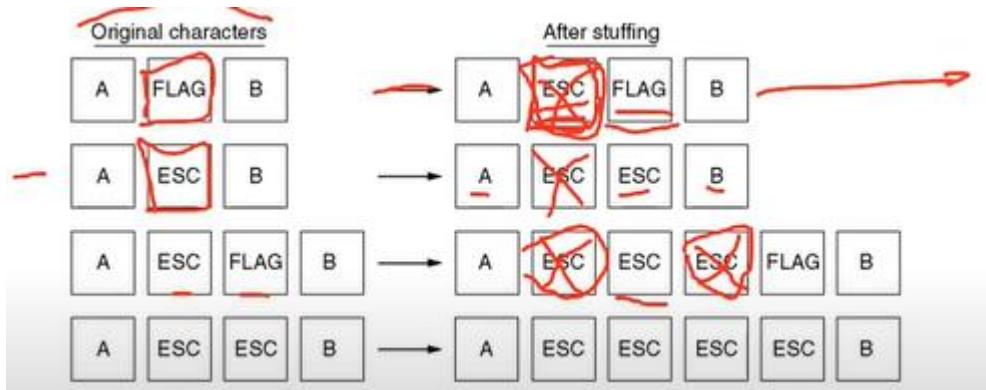


- **Flag byte →**



Se agrega un conjunto de bits constantes (FLAG), delante y detrás de la trama, para indicar inicio y cierre.

En el caso de q tengamos un byte de flag dentro de el Payload, y no se interprete como final de trama: (se agrega 1 byte de ESC, escape.)



Y si además, hubise 1 byte igual al del escape en payload, también se le agrega otro byte de escape delante. (2do caso).

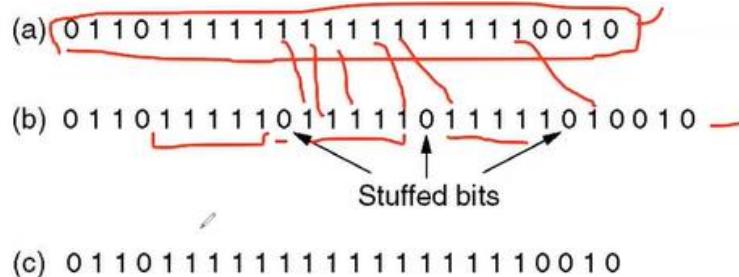
Y así para los demás casos.

En caso de **error**, no queda desincronizado como en el método anterior.

Este protocolo es menos eficiente con la cantidad de bytes de control q hay q mandar.

- **Flag Bit→**

Se agrega al inicio y al final: 01111110 (0x7E)



(a): es el payload

(b): En el tramo se van contando la cantidad de 1 q hay, si se cuentan 5 el siguiente 1(si lo hay) se lo pone en 0, para q ese byte no sea = a los bits de FLAG.

(c): Es la trama q sale de la capa de enlace del receptor (la q llega a la capa de red), es decir es la misma info q se quería enviar originalmente.

→Este modelo a diferencia del anterior **es + eficiente** porq solo se agregan bits, no bytes como el anterior.

- **Physical Layer Violations** →

Se utiliza para transmitir combinaciones de bits invalidas para los datos.

Ejemplo: 4B/5B, es decir en datos de 4 bits, se codifican y envían datos de 5bits, por lo q van a existir combinaciones de 5bits de codificación, q no van a ser validas en datos de 4 bits de la información real, por lo q esas **combinaciones invalidas** se utilizan como **inicio o fin** de trama.

Data (4B)	Codeword (5B)	Data (4B)	Codeword (5B)
0000	11110	1000	10010
0001	01001	1001	10011
0010	10100	1010	10110
0011	10101	1011	10111
0100	01010	1100	11010
0101	01011	1101	11011
0110	01110	1110	11100
0111	01111	1111	11101

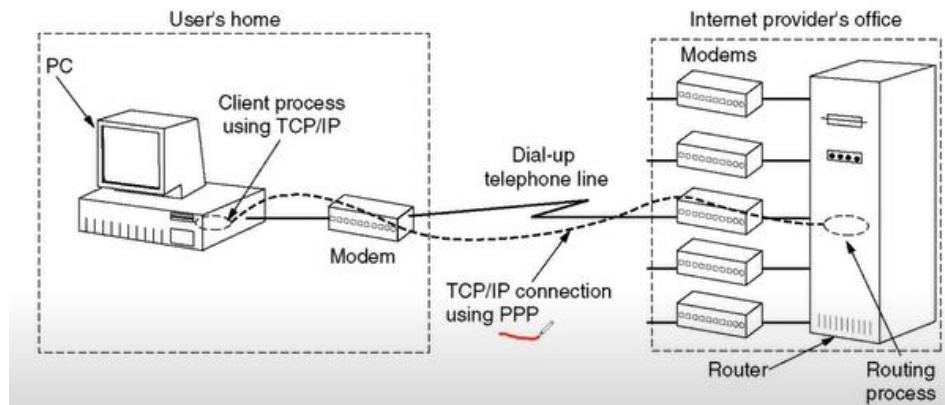
## **Control de errores:**

- Sin confirmación y sin conexión → lo ignora al control (No tiene)
- Con confirmación y sin conexión →
  - 1 frame de control como feedback
  - Con 1 Timmer (se espera n veces hasta q conteste, si no F)
  - El problema de este, q se pueden generar Tramas Duplicadas, es decir q se envia 1 trama A, y el receptor envia el acuse de recibo, pero este se pierde, por lo q no llega al emisor, entonces envia nuevamente la trama A. El receptor no interpreta si es la misma trama o q, solamente la agrega, por lo tanto obtiene la misma trama duplicada.
- Con confirmación y con conexión → Las tramas se enumeran, y sus acuses de recibo. Este es – eficiente en tiempo y canal.

## **Detección y corrección de errores:**

- Código de corrección de errores:
  - Para canales muy ruidosos
  - Se añade siempre info de control
    - Hamming
    - Convolucional binario
    - Reed-Solomon
    - Código de chequeo de paridad de baja densidad
- Códigos de detección de errores:
  - Para canales no tan ruidosos
  - Por lo q es necesario solamente detectar los errores nomas, y retransmitir esa info.
    - Paridad
    - Checksum
    - Cyclic Redundancy Checks (CRC)

→ 1 caso de uso de 1 protocolo de capa de enlace (protocolo PPP- punto a punto protocolo):



Utiliza el 0x7E:

Bytes	1	1	1	1 or 2	Variable	2 or 4	1
	Flag 01111110	Address 11111111	Control 00000011	Protocol	Payload }}	Checksum	Flag 01111110

El **checksum**, es para sumar **paridad** y detectar errores.

## Clase 15: Sub Capa Mac

**The Medium Access Control:**

Para redes punto a punto, NO son necesarias. Pero para redes de Broadcast SI

Ya que es 1 canal compartido (1 frecuencia o 1 cable compartido, etc), tengo varios nodos.

Por lo tanto, **Cómo accedo** al canal se llama: **Medium Access Control**.

Maneras de acceder al canal:

- **Asignación Estática in LAN y MACs:**
  - Se divide en frecuencias (rara vez usado).
  - FDM y TDM
  - Se necesita 1 número fijo de nodos
  - Los nodos deberían transmitir todo el tiempo (mucho tráfico)
- **Asignación Dinámica in LAN y MACs:**
  - Todas las estaciones q van a transmitir son independientes

- Hay 1 solo canal q va a estar compartidos por todos los nodos. Y todos estos **tienen = capacidad de transmisión**.
- Se pueden observar las **colisiones**. Cuando hay 2 o + transmisiones en simultaneo. Hay q retransmitir los datos.
- Puede ser 1 tiempo continuo o ranurado.
  - Si es Continuo, los datos pueden mandarse cuando estén
  - Si es Ranurado, deben ser al inicio de esa ranura de tiempo.
- Sensado de Portadora o No:
  - Si sensan portadora, los nodos podrían saber si hay actividad en el canal.

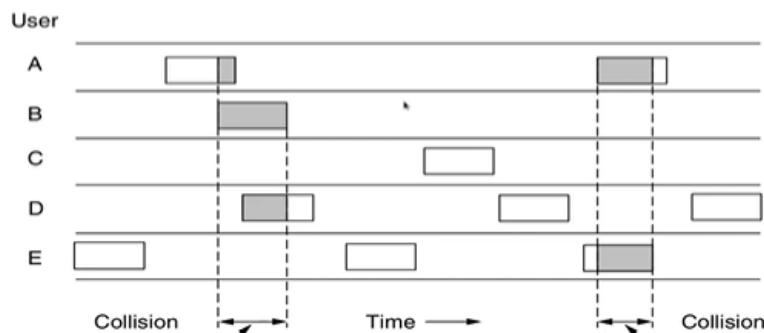
## Protocolos:

- Aloha
- CSMA (Carrier Sense Multiple Access Protocols)
- Protocolo libre de colisión
- Protocolo de contención limitada
- Protocolo Wireless LAN

### Aloha (Inalámbrico):

1 nodo transmite, y el Nodo Central recibe lo q transmitió el nodo y lo retransmite, así el nodo q transmitió se entera de q su dato fue transmitido con éxito.

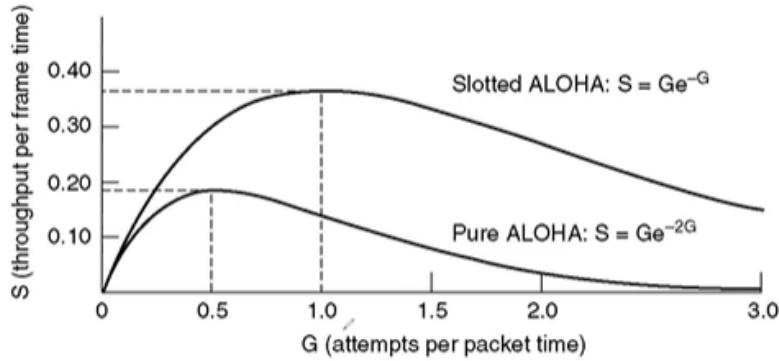
En el caso de colisión de datos, cada nodo espera 1 tiempo aleatorio, ya q si por ejemplo le defino 1 segundo para q retransmitan, van a volver a colisionar y así sucesivamente.



### Aloha Ranurado:

Es 1 mejora del anterior. Los nodos ahora no transmiten cuando tengan el dato, si no q esperar a q se inicie la ranura de tiempo.

Para sincronizar las estaciones, la Central tira 1 trama muy corta (BICON?), q cuando los nodos q tengan dato reciban esta trama ahí recién transmiten. (Puede existir colisión cuando 2 o + reciben la trama, pero no cuando 1 está transmitiendo, como ocurría en el anterior.)



Throughput versus offered traffic for ALOHA systems.

$$G = N + Ret$$

$$S = P_0 G$$

(Eje horizontal → Cantidad de tramas totales a transmitir, las q hay q transmitir + las q hay q retransmitir). (**INTENTOS**)

(Eje vertical → Eficiencia en el canal.) (**RENDIMIENTO**)

Es decir, q en el Ranurado se logra una > eficiencia q en el ALOHA puro.

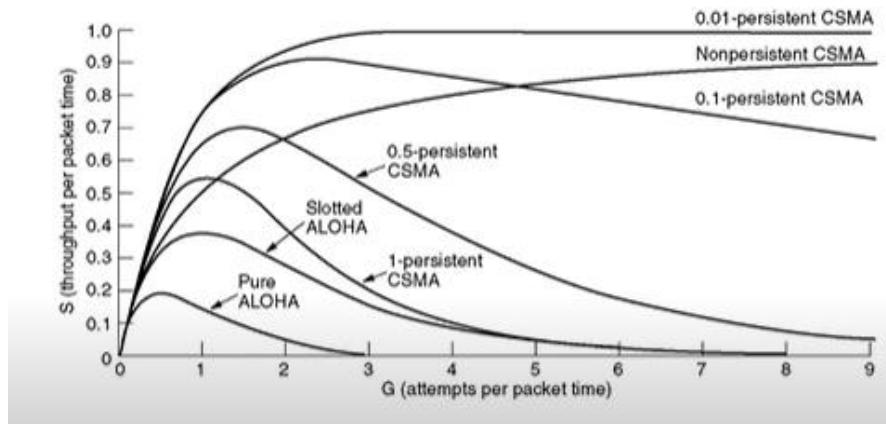
### **CSMA** (Multiple acceso y sensado de portadora):

**Carrier Sense** → Porq antes de transmitir 1 dato me fijo si alguien esta transmitiendo. (En ALOHA esto NO se podía hacer, debido a q los nodos no tenían alcance unos a otros, solamente a algunos y al NODO CENTRAL. Es decir q no tienen forma de saber si están transmitiendo o no, porq se atenúa la señal)

**Multiple Acces** → Porq todos los nodos tienen acceso al mismo bus.

- **Persistente 1 CSMA:**
  - Nodos esperan hasta q nadie este transmitiendo por el bus, para poder transmitir el dato
  - Puede pasar q justo 2 nodos q tienen dato, estaban esperando y justo se desocupa el bus, y quieren transmitir → **Colisión**.
- **No Persistente CSMA:**
  - Escuchan el canal, si no hay nadie, transmite.
  - Si hay alguien, No espera, espera 1 Tiempo Aleatorio. Si siguen transmitiendo, vuelve a esperar otro tiempo aleatorio y así.
  - Se evitan muchísimas colisiones, pero tienen > Delay
- **Persistente P CSMA:**
  - Se aplica para canales ranurados
  - Si tienen datos, **esperan 1 inicio de trama**

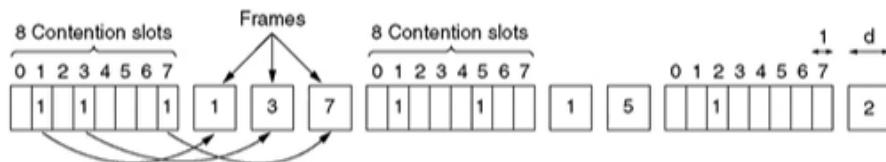
- Van a transmitir esa trama, de acuerdo con 1 **probabilidad P**.
- La + común es Persistente 0.5, es decir q si tienen datos, van a transmitir con un 50% de probabilidades.



(Cuando tengo + paquetes para transmitir, el rendimiento del canal disminuye. La diferencia entre protocolos es q lo hacen en menor medida q otros).

### Protocolo libre de colisión:

- **Protocolo de Mapa de Bits:**



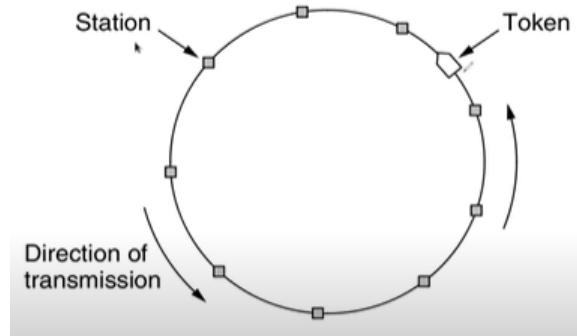
(Con N=8. 8 estaciones quieren usar el mismo medio)

Cada nodo si tienen dato a transmitir, ponen en 1 en la ranura q le toca a cada uno en esa trama. Por lo tanto al principio van a transmitir el 1, el 3 y el 7.

En la 2da trama, va a transmitir el 1 y el 5. Y así...

Este protocolo NO escalable, ya q si hay muchos nodos, se hace 1 poco inviable el método, ya q se hace muy grande la trama de control.

- **Token Passing (Paso de ficha, o paso de token):**



Los nodos están interconectados como se muestra.

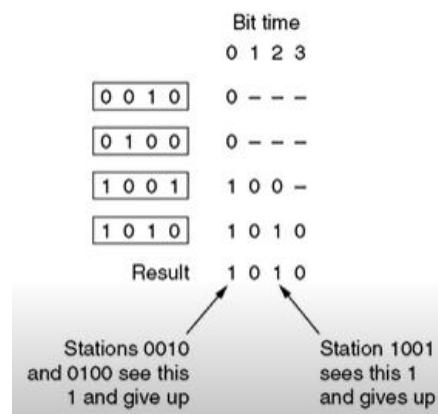
**Token** → es 1 trama o patron establecido chico.

1 estación **No** puede enviar 1 dato, hasta q no le llegue el TOKEN.

No hay colisiones.

Esto NO escala. Ya q si hay muchos nodos, se hace muy larga la espera.

- **Conteo de segmento binario:**



A cada nodo se le asigna 1 dirección. En el ejemplo de la imagen, se tienen 16 nodos (4 bits), cada uno tiene 1 dirección asignada.

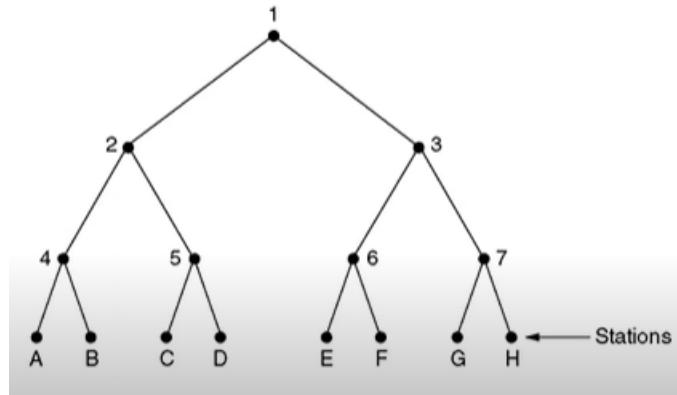
En ese ejemplo, están intentando transmitir 4 nodos. Se empieza a comparar el primer bit de la izquierda, los q tienen 1 siguen en la espera, los demás no pueden transmitir. Por lo tanto, los últimos dos nodos pueden seguir, los primeros dos no. En el tercer bit, como el ultimo nodo tiene un 1, puede seguir. Por lo tanto, termina transmitiendo ese último nodo.

#### CSMA vs Protocolos Libres de Colisión:

- Cuando hay Poca Carga → Son mejores los CSMA (-Delay)
- Cuando hay Mucha Carga → Son mejores los Libre Colisión.

### Protocolo de contención limitada:

- **Adaptive Tree Walk Protocol** (algo así como protocolo de paso adaptativo): se plantea en este protocolo, de la diferencia echo anteriormente, q cuando exista poca carga funcione como 1 CSMA. Y cuando exista mucha carga, funcione como Libre de Colisión.



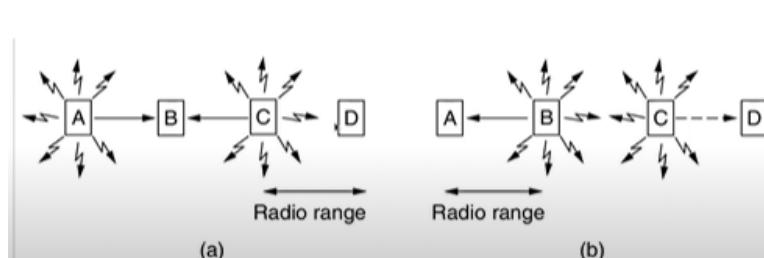
Si hay muchos q quieren transmitir, x ej: el B,D,F y G , entonces va a haber competencia por usar el canal. Por lo tanto 1, se va a fijar primero los nodos q quieren transmitir debajo de 2, y el 2 a su vez, se fija debajo primero de 4 y después de 5. Y así el resto.

Si hay pocos q quieren transmitir, entonces se los deja q transmitan, por lo q NO se va bajando por las ramas, q estas hacen de límite de hosts q pueden transmitir.

### Protocolos Wireless LAN (redes inalámbricas):

No puedo aplicar sentido de portadora (CSMA), porq puede pasar q 1 nodo este lejos de mi alcance y no puedo sensar si esta transmitiendo o no. (Problema de Terminal Oculta-

**Caso (a)):**

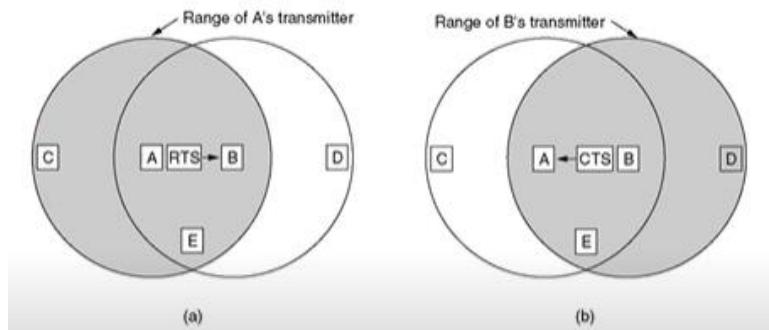


Caso (a) → **Terminal Oculta**. A y C le transmiten a la vez a B, ninguna de las 2 se enteran q están transmitiendo en el mismo momento, porq sus rangos no se logran detectar.

Caso (b) → **Terminal Expuesta**. B quiere transmitirle a A, y C a D, pero B y C se chocan, por lo q deciden no transmitir, debido a q “creen”, q ya se les esta transmitiendo a sus receptores.

X Esto no puede sensar portadora inalámbricamente de esta manera.

- **MACA:**

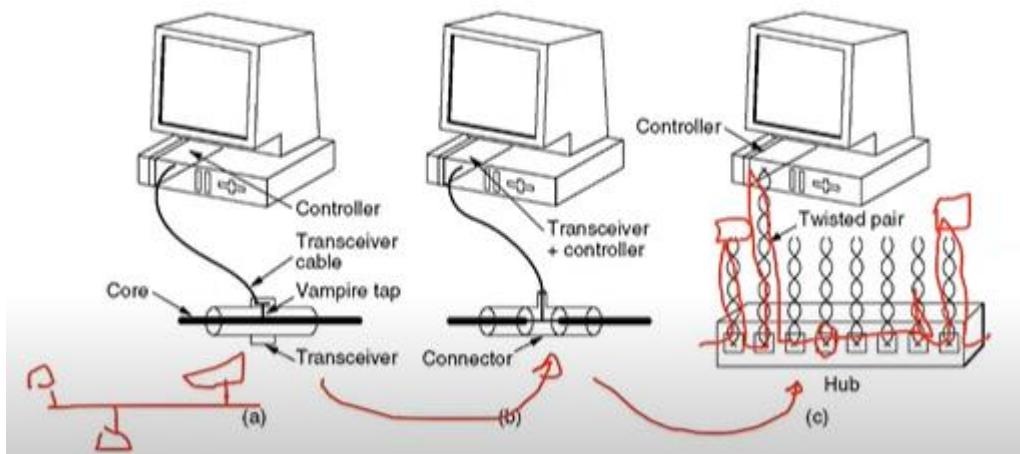


(a) → A quiere transmitir a B, envia 1 **RTS** (ready to send), por lo q esto lo recibe el C, E y B.

(b) → B le responde a A enviando 1 **CTS** (clear to send), por lo q lo recibe E q ya sabía, y D también ahora.

Por lo tanto, los q se enteran esperan, a q A envie los datos a B.

## Ethernet:



(a) → 10 Base 5

(b) → 10 Base 2

(c) → 10 Base T (par trenzado)

Trama (Estamos en **capa de enlace**):

Bytes	8	6	6	2	0-1500	0-46	4	
(a)	Preamble	Destination address	Source address	Type	Data ..	Pad	Check-sum	
(b)	Preamble	SOF	Destination address	Source address	Length	Data ..	Pad	Check-sum

(a) → DIX Ethernet

(b) → IEEE 802.3 . Lo anterior pero normalizado. Se agrego el byte de SOF, y se cambio el Tipo por la Longitud. Este Dentro de Data, tiene unos bytes LLC para definir el Tipo.

Estos 2 pueden convivir en simultaneo. Se diferencian 1 del otro, ya q en (a), se fijan que el valor de Type sea > de 1500, si es así es 1 trama de Ethernet, si no de IEEE, ya q se estaría especificando 1 Largo.

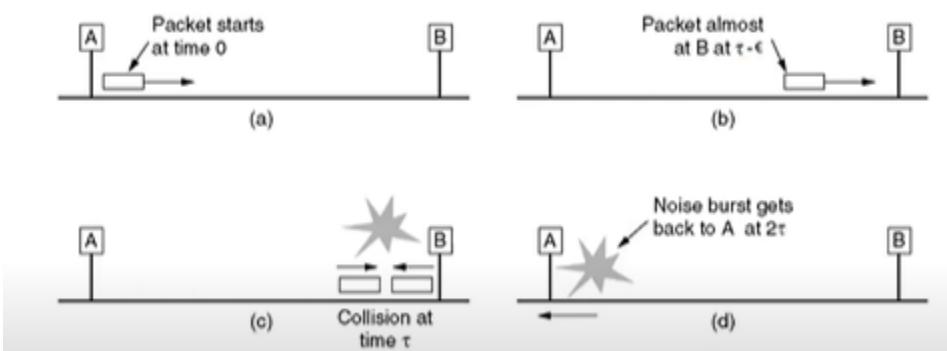
Source → Porq al q le llega la trama, sepa de donde vino y a quien tiene q responderle.

Destination → Para saber a quien hay q enviararse. Además para los demás nodos q estén conectados a la red, sepan q No es para ellos el mensaje. (Direccion de **MAC**)

Type → Para saber q tipo de protocolo de red es, si es IP o q , los datos (Data + Pad) quiero enviar.

El relleno es para agregarle un valor mínimo de datos, porq por ahí la Data es < a los 46 Bytes. (**Esto se usa para detectar Colisiones**)

Eso último, se calculo para el peor de los casos:

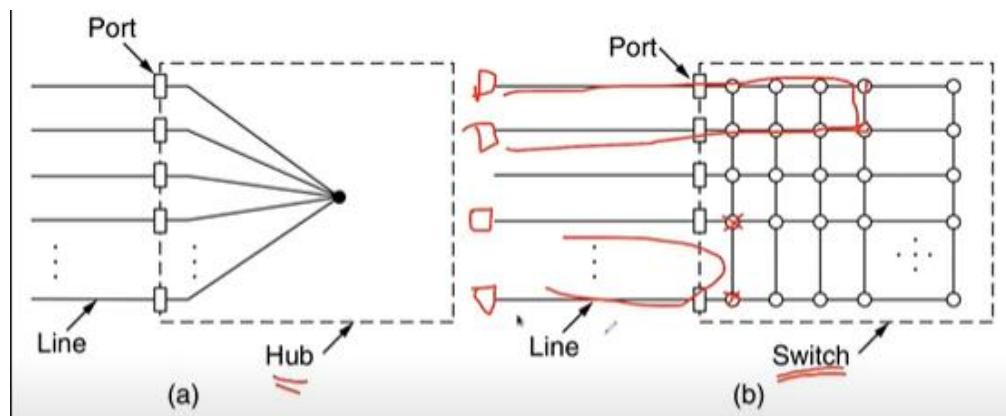


A se da cuenta en 1 tiempo  $2T(\tau)$  hubo 1 colisión. Ocurrió en un instante infinitesimal  $\epsilon$  antes q llegue el mensaje a B.

En 2T debería seguir transmitiendo A. Para q lo q cense de lo q le está llegando es distinto a lo q esta transmitiendo.

Con este dato y la velocidad de propagación a la q se transmite se calcularon los 46 Bytes mínimos. Para q en 2T pueda detectar la colisión. (**En total la trama son 72 Bytes**)

### Switch vs HUB:



**Hub** → Lo que llega por 1 puerto, lo retransmite por todos los demás, por lo q a los demás si alguien estaba enviando datos, van a haber colisiones en esos puertos, pero al q se desea transmitir le va a llegar =.

**Switch** → Solo se transmite el mensaje al destinatario necesario.

Ejemplo del switch:

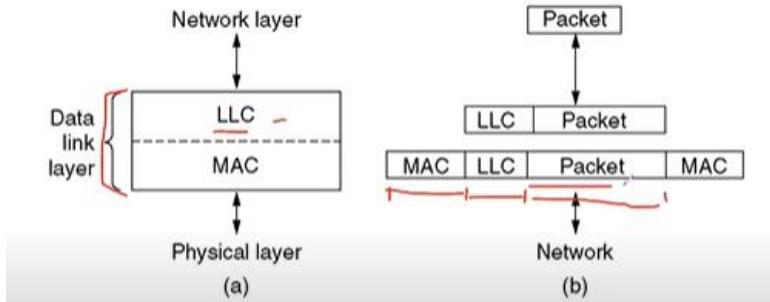
Posee 1 tabla donde switchean 1 puerto con la dirección MAC del destino. El puerto 3 se switchea para enviar desde el puerto 3 (nodo A), 1 dato al puerto 5 (nodo B)



(Tabla q asocia dinamicamente puertos)

En el caso de q 1 MAC no este asociada en la tabla con algun puerto, lo q hace si 1 puerto se quiere conectar con este, y no esta en la tabla, se envia a todos los puertos el mensaje, hasta q el puerto requerido contesta. Ahí recien se agrega la dirección MAC al puerto donde se encuentra

### Sub Capa LLC:



Es 1 sub capa de la sub capa MAC. Para detallar el **Tipo**. Es para ver a **q tipo** (Protocolo q tiene) **de capa de Red** se le envía la trama.

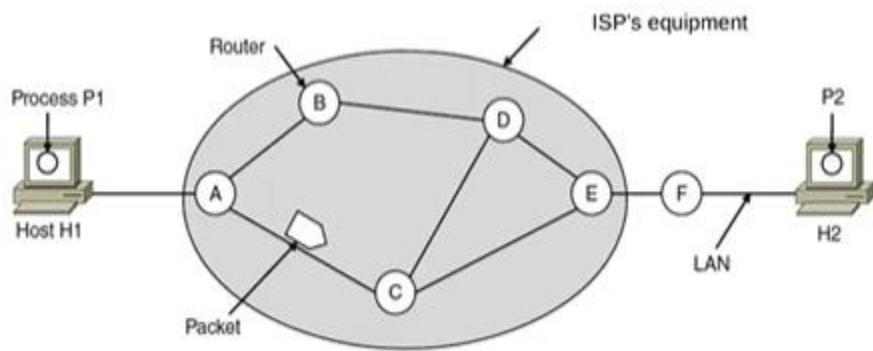
## Clase 16: Capa de Red

Es la capa q se encarga de enviar y recibir paquetes entre los nodos.

### Diseños:

- **Store and Forward packet switching** (ruteo y encaminamiento).
- **Servicios Sin Conexión.**
- **Servicios Con Conexión.**

### Store and Forward packet switching:



Los **Hosts** envían paquetes a través de la red, y los **Routers** los encaminan.

Los routers, almacenan toda la info q le va llegando del paquete, para después procesarla y enviarla en alguna ruta.

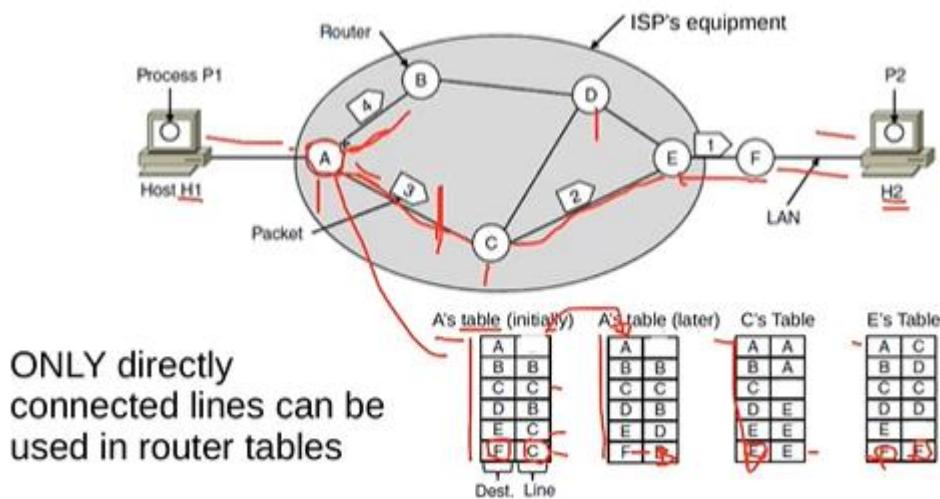
## Consideraciones →

- El servicio q da esta capa es Independiente de la tecnología que tiene la capa de enlace. (PPP, 802.11, 802.3, etc)
  - Y la capa superior a esta, La Capa de Transporte, también no debería interesarle la cantidad de Routers q se atraviesan, la cantidad de redes q se atraviesan, etc. (**Ese es el servicio q ofrece la capa de red. La capa superior se tiene q olvidar de todos los caminos q tengo q ir**)
  - Esta capa debe tener **1 Identificación Única y Uniforme** entre todas las maquinas. Para poder abstraerme de las distintas tecnologías de la capa de enlace.
  - Puede ser Orientada a Conexión (Similar al servicio de Telefonía de celular) o NO Orientada a Conexión (Similar al servicio de Internet).

## Servicios Sin Conexión:

(Datagramas o Paquetes).

Que sea sin conexión significa que manda los datos, sin establecer 1 conexión previa.

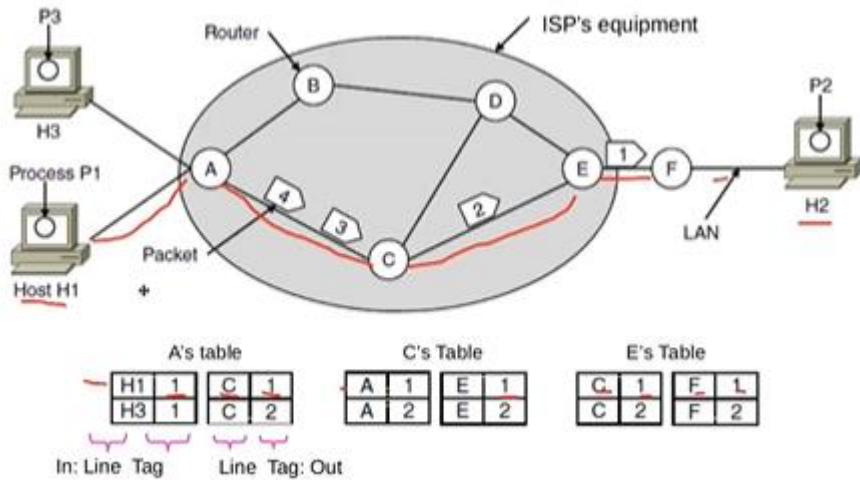


Si el host1 quiere enviarle 1 paquete al 2, 1ero el router A, se fija q tiene q llegar a F, por lo tanto, se fija en su tabla dinámica q tiene q enviarlo por C, (en el caso q se rompa el enlace con C, se fija en la “tabla prima” que debe enviar hacia F por B), y así sucesivamente van haciendo los siguientes routers.

Es decir, q 1 paquete puede tener distintos caminos.

#### **Servicios Con Conexión – Circuitos Virtuales:**

Lo 1ero q se hace, antes de enviar 1 solo paquete es, crear un circuito virtual, es decir, q a la información q quiero enviar se le pone 1 etiqueta del número de circuito que es. La etiqueta es **MPLS** (20 bits)



Si se corta un enlace, hay q rehacer el circuito virtual o esperar q se vuelva a conectar el enlace.

**Ventajas**→ Es + simple y rápido la transmisión de paquetes. Puedo identificar por donde está yendo la información. En 1 mismo router puedo asignar cierto ancho de banda a 1 circuito virtual y otra cierta cantidad a otro. (**Asegura Calidad De Servicio**)

**Desventajas**→ EL problema de q si se corta el enlace, hay q rehacer el circuito virtual, en cambio el anterior lo hace solo a la retransmisión.

Comparativa:

Issue	Datagram network	Virtual-circuit network
Circuit setup	Not needed	Required
Addressing	Each packet contains the full source and destination address	Each packet contains a short VC number
State information	Routers do not hold state information about connections	Each VC requires router table space per connection
Routing	Each packet is routed independently	Route chosen when VC is set up; all packets follow it
Effect of router failures	None, except for packets lost during the crash	All VCs that passed through the failed router are terminated
Quality of service	Difficult	Easy if enough resources can be allocated in advance for each VC
Congestion control	Difficult	Easy if enough resources can be allocated in advance for each VC

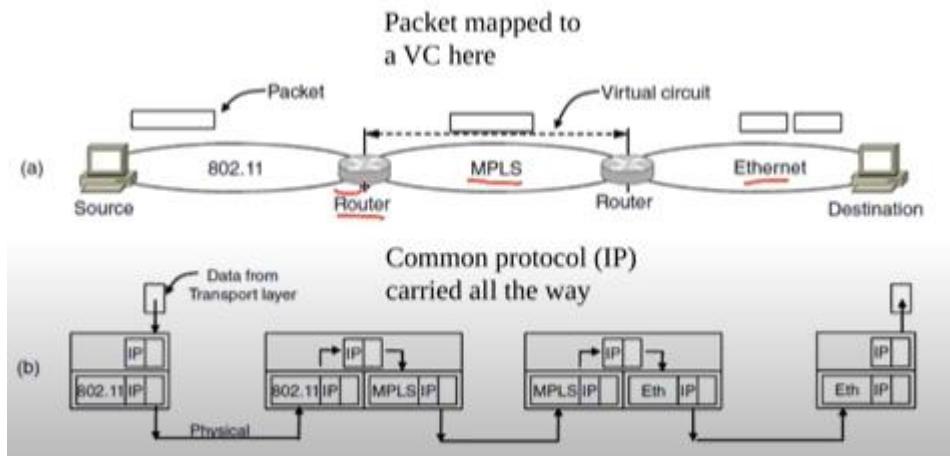
## Internetworking:

Inter conectar distintas redes de capa de enlace, como si fuera 1 red + grande

Como difieren las redes:

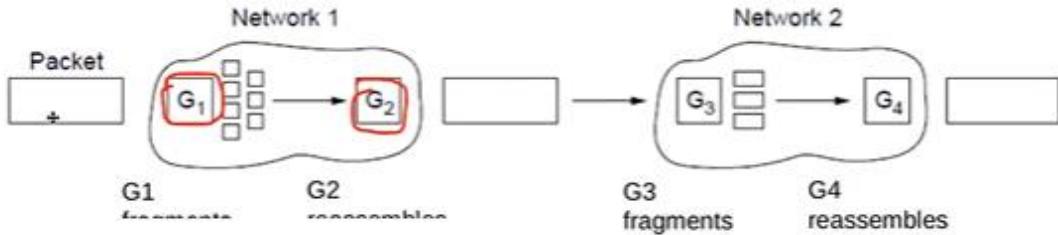
Item	Some Possibilities
Service offered	Connectionless versus connection oriented
Addressing	Different sizes, flat or hierarchical
Broadcasting	Present or absent (also multicast)
Packet size	Every network has its own maximum
Ordering	Ordered and unordered delivery
Quality of service	Present or absent; many different kinds
Reliability	Different levels of loss
Security	Privacy rules, encryption, etc.
Parameters	Different timeouts, flow specifications, etc.
Accounting	By connect time, packet, byte, or not at all

Ejemplo de 3 redes:

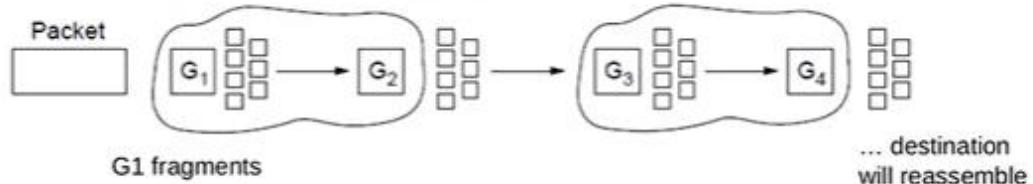


## Packet Fragmentation:

Cada red (+ específicamente la capa de enlace no soporta) soporta 1 tamaño específico de paquetes. Por lo q hay veces q es necesario fragmentarlos:



Transparent – packets fragmented / reassembled in each network

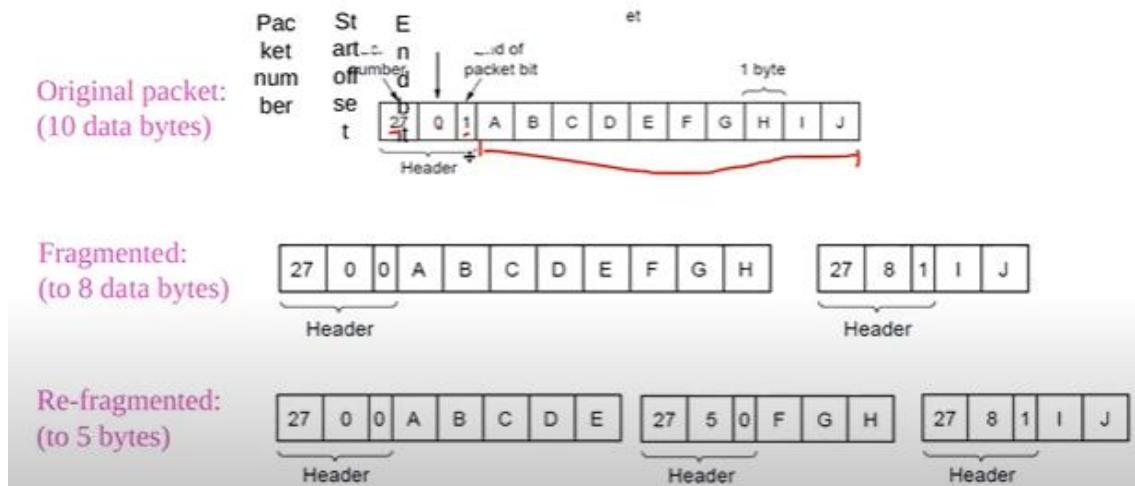


Non-transparent – fragments are reassembled at destination

**Transparente** → Porq el router q le llega fragmentado lo reensambla al paquete. Es decir q el destino ni se entera q fue fragmentado.

**No Transparente** → No se reensambla en los routers subsiguientes. Lo reensambla la máquina de destino nomas.

Ejemplo de fragmentación:



Paquete original:

- Número de paquete → 27
- Offset → 0
- Bit q se termina el paquete → 1

Fragmentado:

- 1er Fragmento

- 27 porq es el mismo paquete pero fragmentado
- Offset:0
- Bit: 0 (porq sigue)
- 2do Fragmento:
  - 27
  - Offset: 8, porq el fragmento anterior tiene 8 bytes.
  - Bit: 1 porq termina

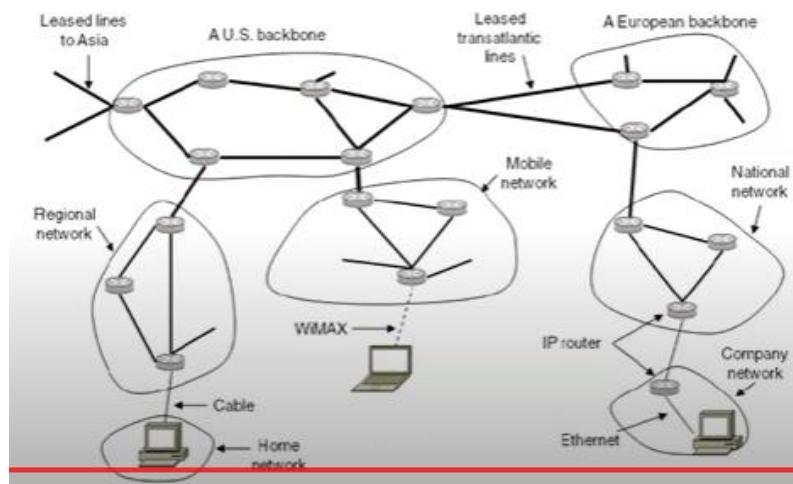
## IPv4:

Se diseño para q el protocolo sea:

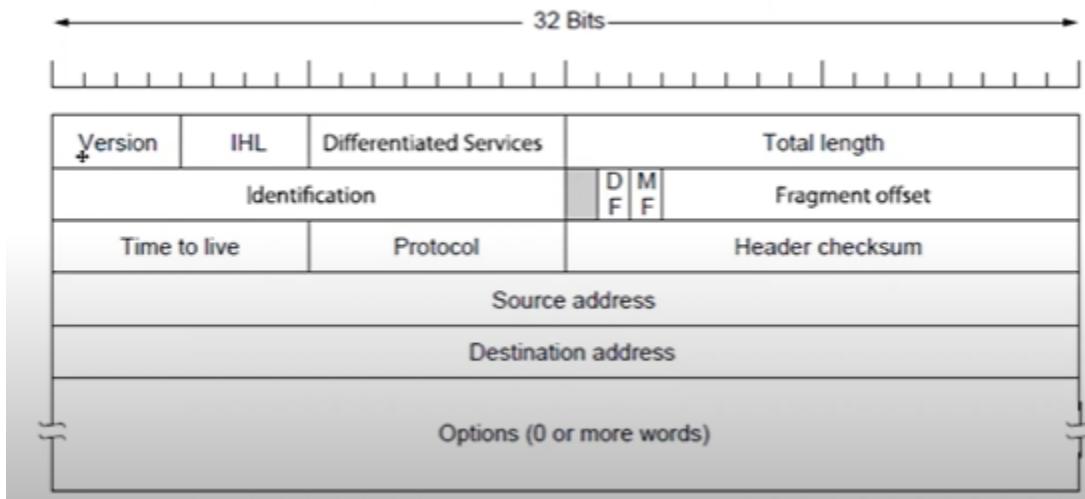
- Sencillo
- Asegurando q trabaje
- Q las opciones sean claras
- Q sea modular y heterogéneo
- Evitar parámetros y opciones estáticas
- Q sea eficiente transmitiendo y tolerante cuando estoy recibiendo
- Q tenga en cuenta la escalabilidad
- Performance y costo

## Internet:

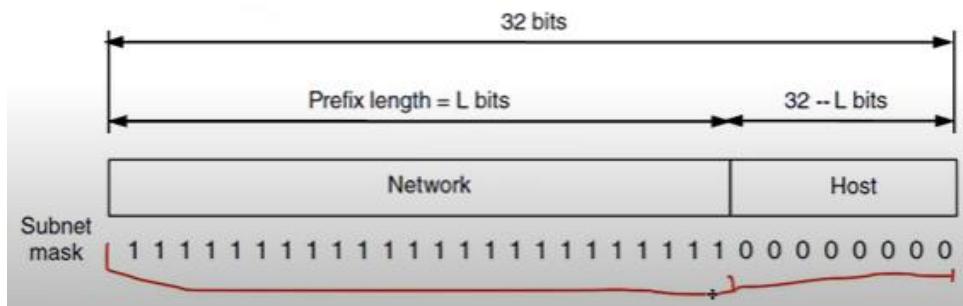
Es 1 interconexión de distintas redes, utilizando el mismo protocolo IP.



Protocolo:



En las direcciones de 32 bits, se especifica en q red están y q host son:



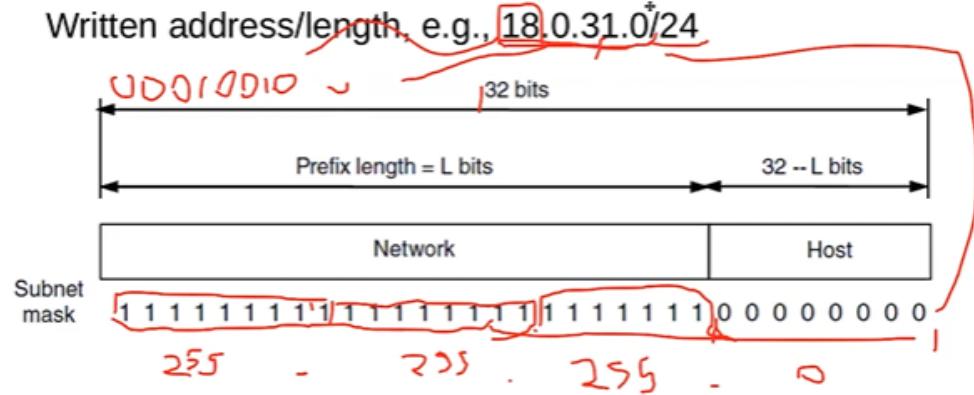
Para hacer el trabajo + sencillo al router, para redirigir el paquete.

(Analogía → Es como ir de Mendoza a Palermo en Bs As. En la ruta vamos a encontrar carteles para redirigirnos hacia Bs As, no hacia Palermo específicamente, por lo menos hasta q no estemos cerca)

Por lo tanto, para el router es + fácil tener las direcciones de las redes, y no 1 a 1 las direcciones de los hosts. (Serían tablas interminables) (**Host**: tiene  $32 - L$  bits)

Ejemplo de red:

- Written address/length, e.g., 18.0.31.0/24



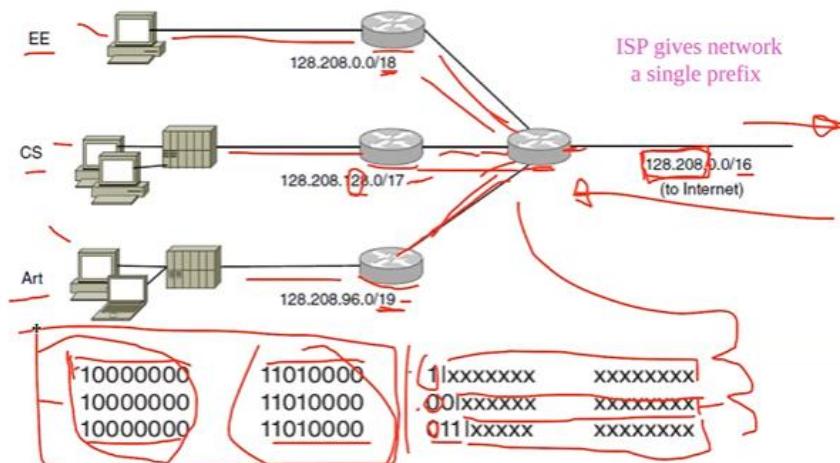
24 → Debido a q son 24 bits de red,  $8 + 8 + 8$ .

**Ventajas:**

- Routers reencaminan los paquetes en función de la dirección de red no de la dirección IP.
- Las tablas son mucho más chicas

**Desventajas:**

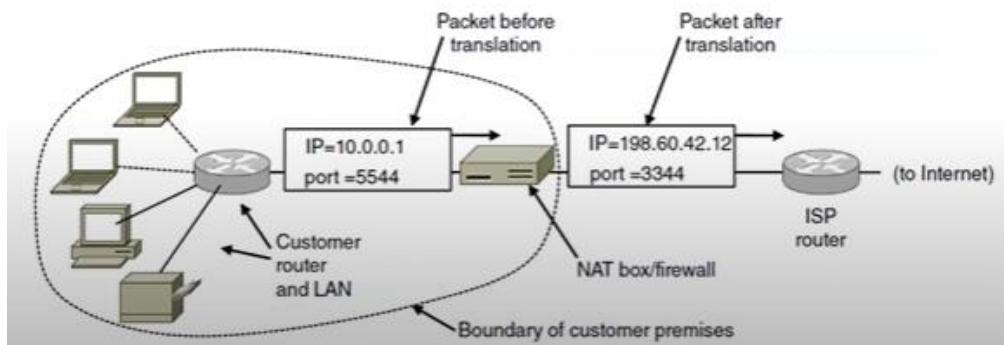
- Todas las redes q tengan la misma parte de red tienen q estar en el mismo lugar físicamente
- Por lo tanto, se desperdician direcciones
- Alternativa → **SubRedes**



## NAT (Network Address Translation):

IPv4 no hay + direcciones disponibles públicas.

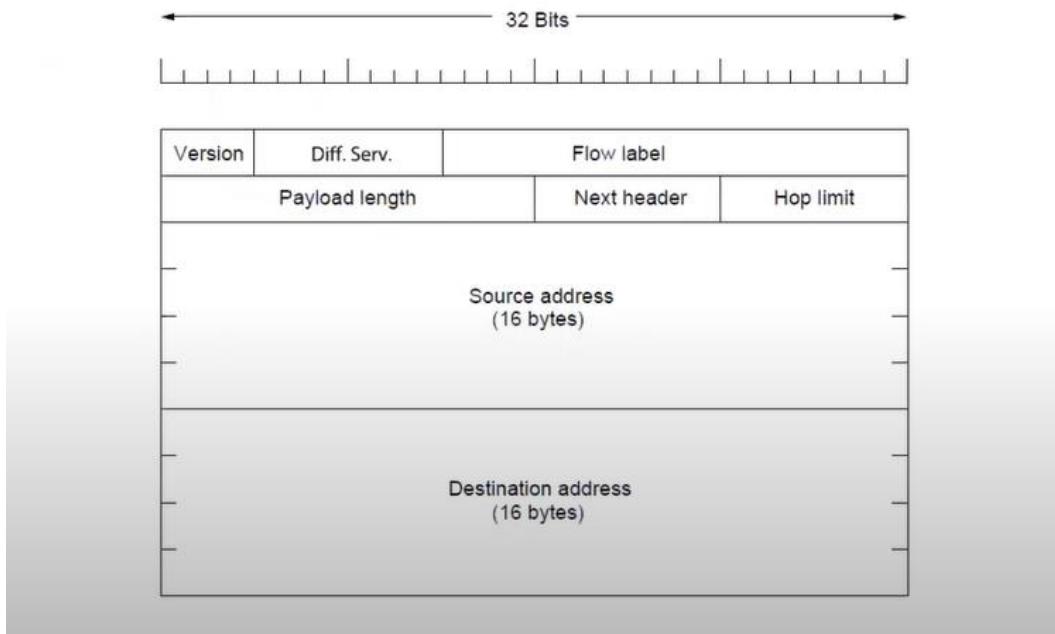
Por lo q se puede poner a 1 red, 1 rango de redes IP privadas, se le cambia la dirección IP, con 1 nueva q le proporciona el router a las distintas maquinas conectadas en esa red privada:



Otra alternativa mejor:

## IPv6:

Protocolo disruptivo. La única forma que convivan los 2 protocolos es que la maquina q use a ambos debe tener los 2 implementados en la pila.



Version → Como es la v6: 0110

Las direcciones son de tamaño: **128** bits (16 bytes)

Tiempo de vida ahora es **Cantidad de Saltos (Hop Limit)**: si se le pone 10, cada vez q pasa por el Router, se disminuye y si llega a 0 se lo descarta.

No esta **Checksum** → Por lo q cuando llegue algo lo decrementa al Hop Limit y no hay q hacer el recalcular del Checksum, por lo q se ahorran esos cálculos. Además, se necesitaba para calcularlo, el **Largo Del Encabezado**, y acá es siempre fijo.

En esta versión **NO** hay **Fragmentación**. Hay 1 mínimo de payload q cada router debería soportar, si no, no es compatible con IPv6.

Los bits de **Diferente Servicio** y de **Flow Label**, en IPv6 prácticamente no se usan, los routers los ignoran.

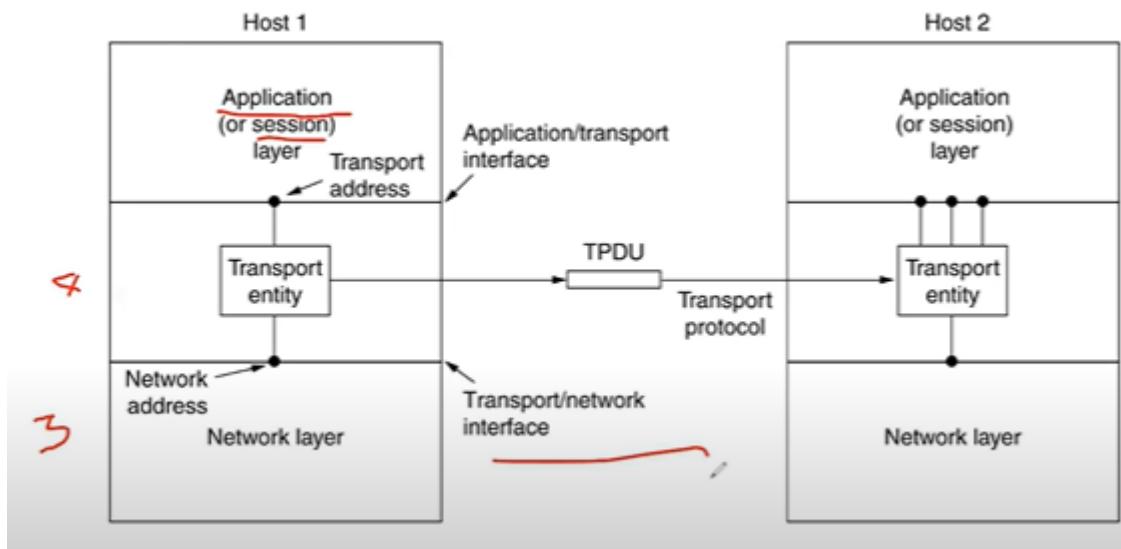
## Clase 17: Capa de Transporte

Existen 2 protocolos en capa de transporte:

**UDP** (User Datagram Protocol): sin conexión

**TCP** (Transport Control Protocol): orientado a acción

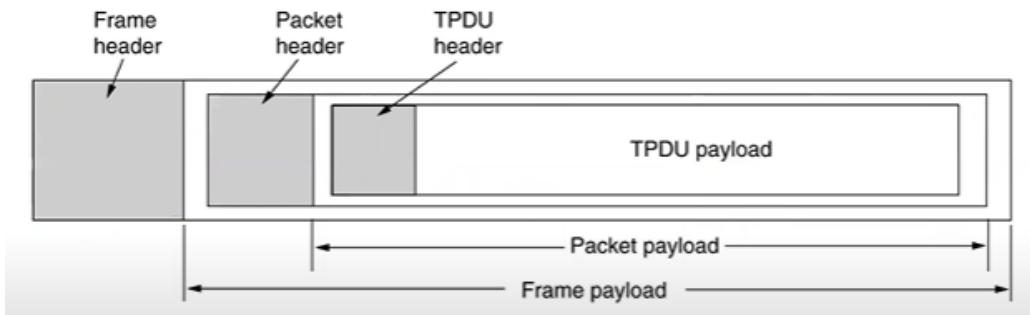
Capa de transporte con su capa superior e inferior:



**Paquetes/Datagramas → Capa Red.** En esta capa el objetivo era la comunicación entre máquinas para enviar info, no importa donde estuvieran, usando 1 mismo protocolo, independizándose de las distintas tecnologías de la capa de enlace

**Segmentos → Capa de Transporte.** Vendría a ser 1 IPC. Ya que el objetivo de esta capa es la **conexión entre el proceso de 1 Host con el proceso de otro Host**. Utilizando 1 nueva dirección, **la dirección de transporte (PUERTOS)**.

Anidado de los distintos protocolos:

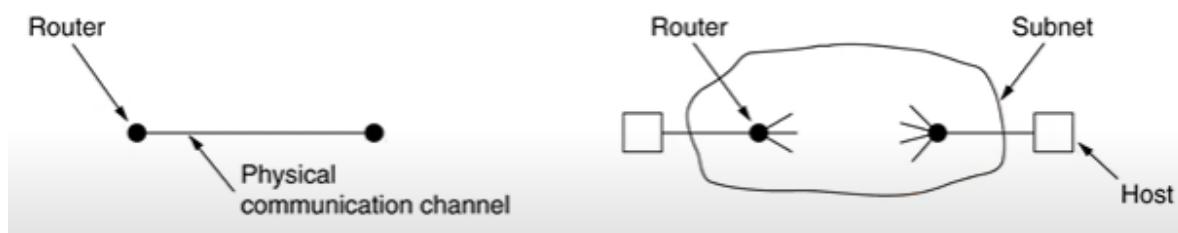


- Frame → Capa enlace
- Packet → Capa red
- TPDU → Capa transporte

**Elementos** de la capa de transporte:

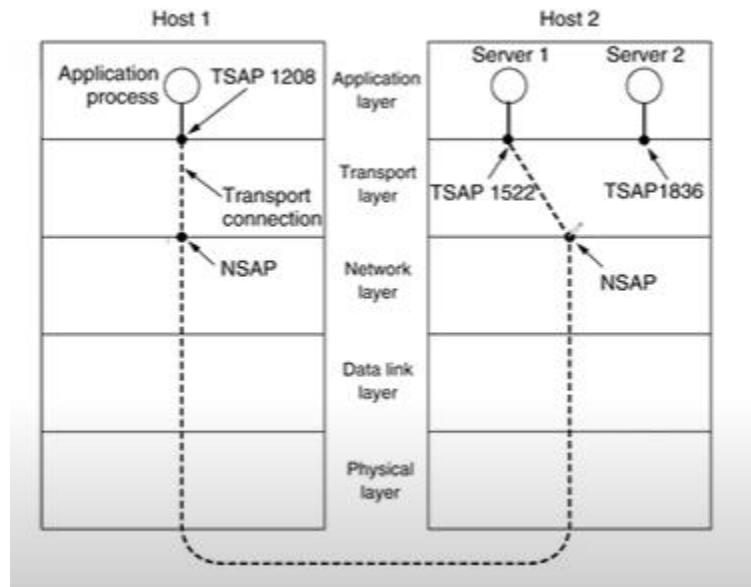
- 1 esquema de Direcciones de transporte
- Si es con conexión:
  - 1 mecanismo para establecerla
  - 1 mecanismo para liberarla
  - Recuperación de caídas
- Podría tener control de flujo y de buffer
- Multiplexación (Los distintos procesos comparten la misma capa de red)

La capa de red funciona como 1 capa de enlace para la capa de transporte, homogeneizando las distintas tecnologías en 1 mismo protocolo.



(a): Capa de enlace, (b): capa de transporte.

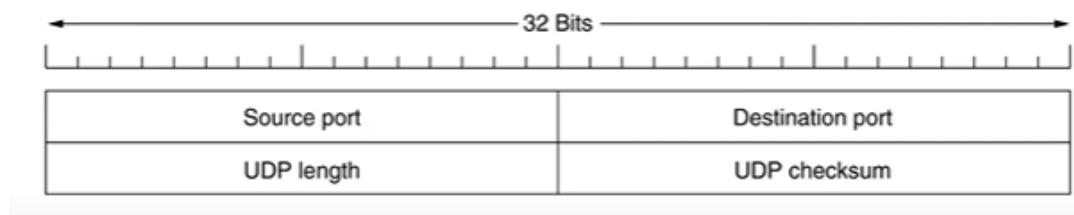
#### Direccionamiento:



En capa de red, teníamos el punto de acceso a la capa de red **NSAP**. Y en capa de transporte tenemos **TSAP** (Transport Service Acces Point), q son números (**puertos**).

- Protocolo Sin Conexión: **UDP**:

Envía datos sin establecer 1 conexión. Si hacia donde estoy enviando los datos es 1 proceso q no existe o se terminó, esos datos se perderán.

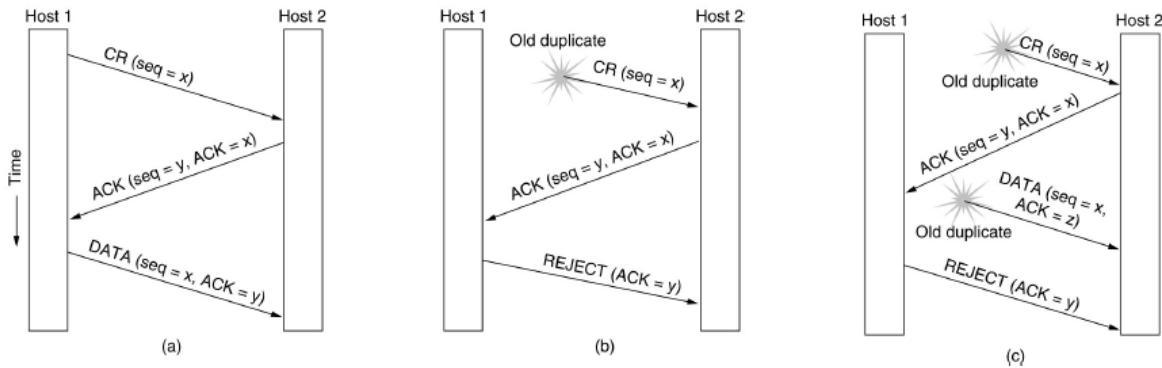


Este protocolo mas q nada envía el puerto de **origen** y **destino**, para poder hacer la multiplexación entre varios procesos la capa de red.

El **checksum** es opcional, podría estar en 0.

Este protocolo es útil para aquellas conexiones por ejemplo 1 en tiempo real, q no importa si hay algunos datos q se pierdan, no hacen falta retransmitirlos.

- Protocolo Con Conexión:



**(a): Operación Normal.** Como son tramas numeradas la información, 1ero se envía la petición de conexión, con el numero de secuencia. 2do el host2 envía el acuse de recibo con su numero de secuencia y el acuse de recibo con el numero de secuencia del host1. (Los números de secuencia son aleatorios, para evitar ataques de seguridad.). 3ero se envían los datos, 1 vez se ponen de acuerdo.

Posibles pérdidas de info y como se recuperan:

**(b): Acuse de recibo viejo duplicado.** El host2 lo responde con la info necesaria. Si el host 1 había enviado la petición hace mucho es posible q lo rechace finalmente, es decir corta la comunicación.

**(c): Acuse de recibo y petición de conexión duplicada.**

## TCP:

**Well Known Ports** (Los puertos bien conocidos):

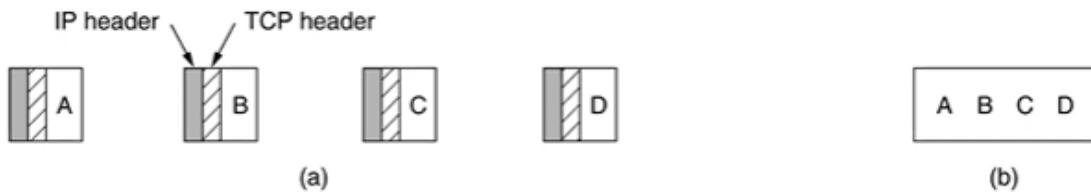
Todos los servidores los asocia siempre al mismo puerto.

Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

X ejemplo asociando solamente el puerto 80 para hacer consultas www, entonces solamente las PCs deben saber la dirección IP del servidor.

1 puerto no puede estar asignado a + de 1 proceso. Porq los datos no van a saber a cuál llegar.

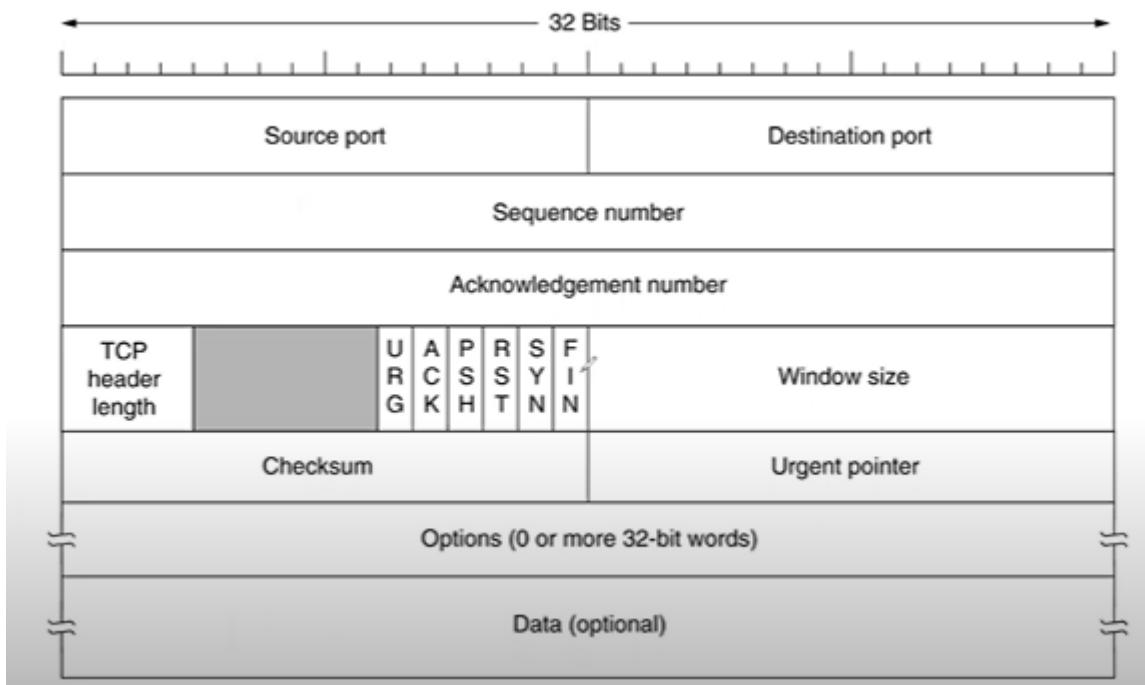
¿Cómo es la comunicación? Ejemplo:



Pueden enviarse 4 segmentos de datos, tal vez pueden llegar en distinto orden, pero **TCP puede ordenarlos a los segmentos**, al contrario de UDP.

Funciona como 1 PIPE BIDIRECCIONAL.

TCP protocolo:



- **Numero de secuencia:** numero q le va asignando a cada segmento
- **Numero de acuse de recibo:** es el numero q se le manda al q envió la información, diciendo cual es el numero de secuencia q se recibió. En realidad, en TCP/IP se envía el numero de secuencia + 1, es decir el numero de secuencia del segmento q se está esperando.

- **Bits:**
  - **SYN**: en 1 para indicar q esta iniciando el saludo de 3 vias
  - **FIN**: para finalizar la conexión
  - **RST**: para el caso q no se haya podido inicializar el saludo de 3 vias.
  - **ACK**: es para indicar q el dato en los bytes de n° de acuse de recibo son validos
  - **URG**: está asociado a los bytes de Urgent Pointer. Sirve para q en vez de procesar los datos como vienen, en los Datos (opcionales), se procese algún dato primeramente, dentro de Datos.
  - **PSH**: bit q si esta activo, es para q no espere llenar el buffer para mandar datos. Si espera, entonces puede enviar varios datos o mensajes en 1 misma trama con 1 solo encabezado.
- **Checksum**: si es obligatorio en TCP.
- **Tamaño de encabezado**: ya que puede ser variable la trama, ya q existe 1 campo opcional q puede llegar a los 32 bits.
- **Tamaño de ventana**: cuando llega 1 acuse de recibo de q lo recibió el destinatario, la trama la puedo sacar del buffer, y puedo seguir transmitiendo + datos. Debido a esto el tamaño de ventana es **variable**, puedo hacer control de flujo con esto. Es decir, q si el receptor no puede procesar todos los datos q le llegan rápidamente, este puede enviar 1 acuse de recibo con tamaño de ventana = 0, lo cual hasta q no se modifique este tamaño, el host transmisor no puede enviar + datos (acuse de recibo SI).

## Clase 18: API Socket

Recordando, había distintos modelos de comunicación:

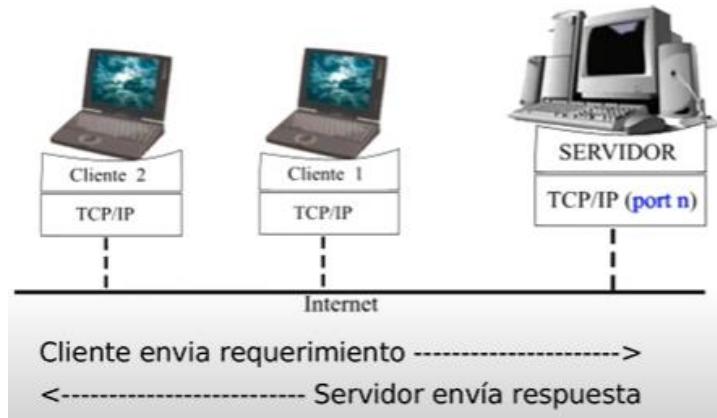
- Modelo Centralizado
  - Problemas de escalabilidad/interoperabilidad
- **Modelo Cliente/Servidor**
  - Flexibilidad / Interoperabilidad / Escalabilidad
- Modelo Peer to Peer
  - Cliente y Servidor simultáneamente
  - Info distribuida / robustez
  - Problemas de seguridad / anonimato

### **Modelo Cliente / Servidor:**

**Servidor** → Aplicación o Proceso, accesible a través de la red, es la q tiene los Datos o Info, esperando requerimientos por parte de los Clientes.

**Cliente** → Aplicación o Proceso, que solicita Info al servidor y espera su respuesta.

Si estos 2 hablan en distintos protocolos, es imposible q se puedan comunicar.



#### Características del Servidor:

- Siempre debe estar ejecutándose
- Utiliza los Well Known Ports
- Utilizan aplicaciones complejas
- Debería atender a + de 1 cliente a la vez.

#### Características del Cliente:

- Se ejecuta solamente cuando va a hacer 1 requerimiento
- Puerto local aleatorio sin usar.
- Debe conocer el IP del servidor (o su nombre).
- Debe conocer el puerto del servidor (o el servicio)
- Inicia la comunicación
- Aplicaciones – complejas.

¿Cómo hacen los programas de aplicación para **usar la red**?

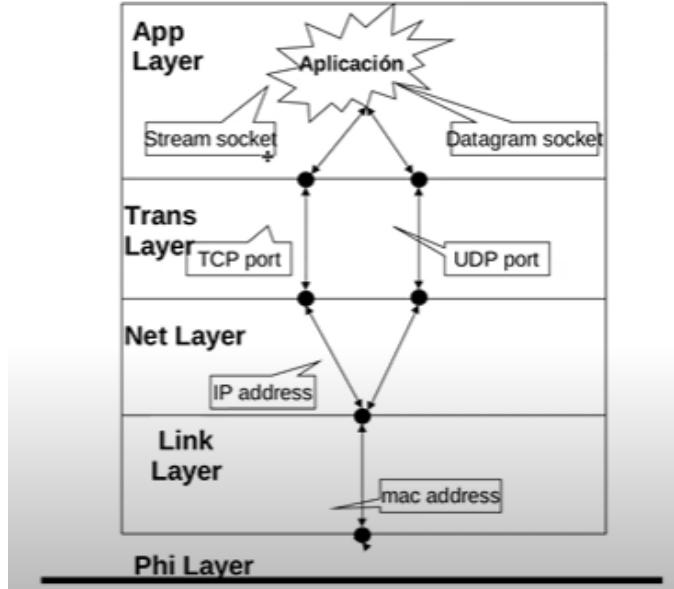
#### Interfaz Socket (enchufe):

El SO implementa toda la parte de Stack de la Capa de enlace, transporte y red. Por lo tanto las aplicaciones corren x fuera del Operativo.

Se necesita de 1 **API** (Application Program Interface), para acceder a los protocolos de red.

Es 1 IPC para procesos no relacionados.

Socket → es adoptado por todos los SO como estándar defacto.



### Características:

- Sigue paradigma UNIX open-read-write-close (TODO es 1 ARCHIVO)
- Usa abstracción de descriptor de archivo
- Uso:
  - Crea 1 socket (fd)
  - Usa funciones específicas de socket
  - 1 vez establecido escribe / lee
  - Cierra Socket

**ENDPOINT** → Se refiere al par: Dirección y Puerto

Socket **Pasivo** → Asociado a 1 solo endpoint (local). Server en estado LISTEN.

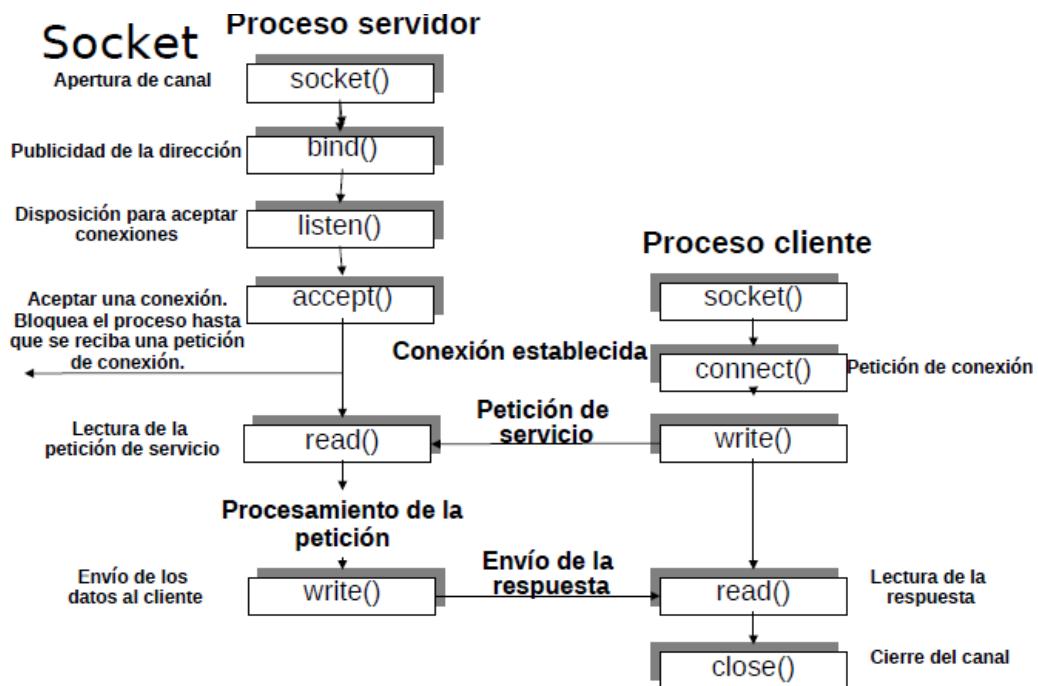
Socket **Activo** → Asociado a 2 endpoints (local y remoto). Para enviar y recibir datos. Cliente y Servidor en estado ESTABLISHED.

### Funciones:

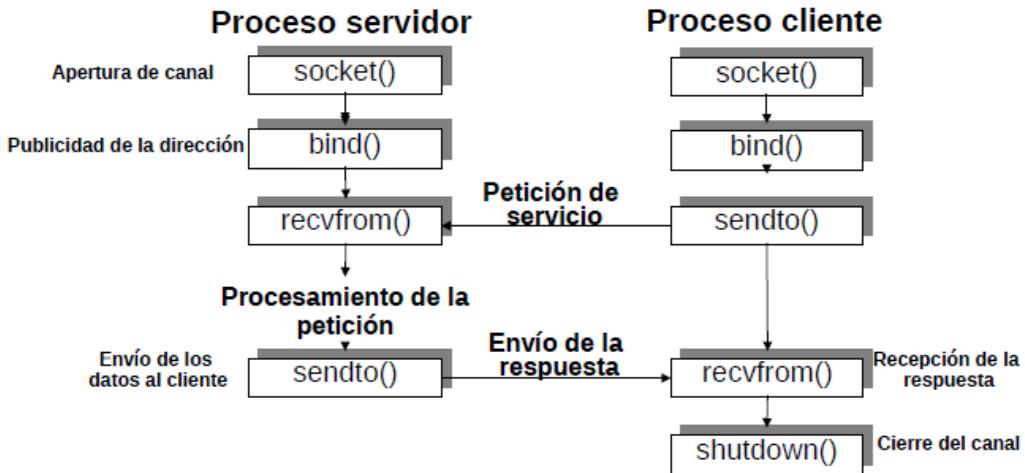
- **Socket ():**
  - Abre 1 canal bidireccional
  - `int socket(int domain, int type, int protocol)`
- **Bind ():**
  - Para asociarlo a puerto local
  - `int bind(int s, const struct sockaddr *name, int namelen)`
- **Listen ():**
  - Crea 1 buffer en el kernel, es decir 1 **Socket Pasivo**. Para q vaya guardando nuevas posibles conexiones, porque no siempre va a poder estar el Socket bloqueado con `Accept()`. Por lo tanto, se perderían esas conexiones.

- int **listen** (int s, int backlog).
- Accept ():  
○ Devuelve 1 **socket activo**.
- Connect ():
- Send ():
- Rec ():

### Modelo Cliente / Servidor – Socket:

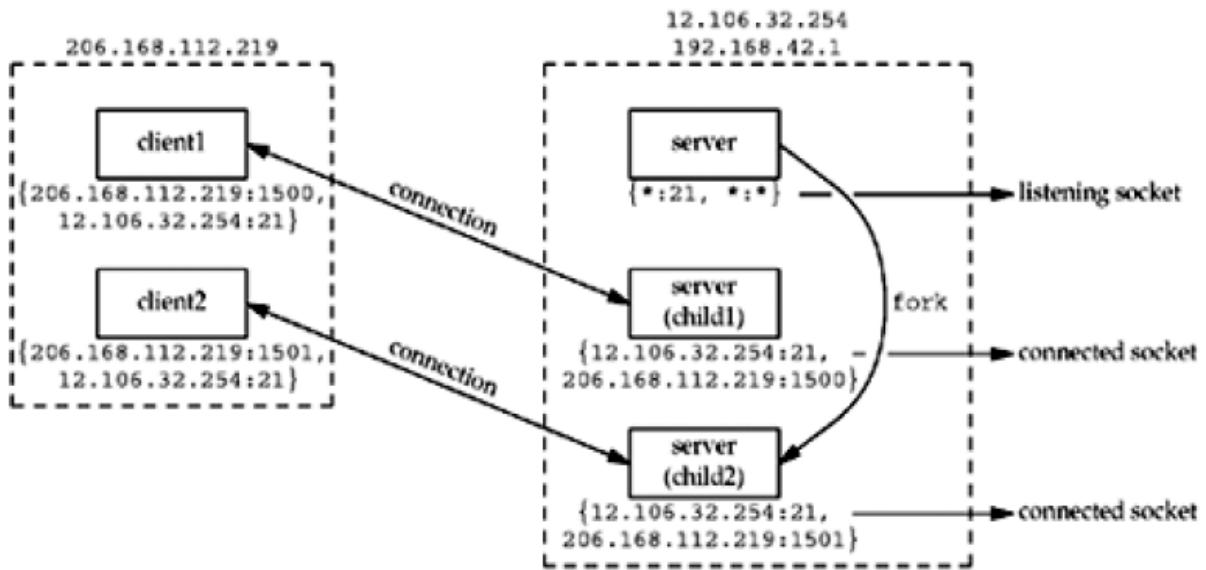


### Socket UDP:



### Con Conexiones Concurrentes:

Se tiene 1 Servidor multiproceso.



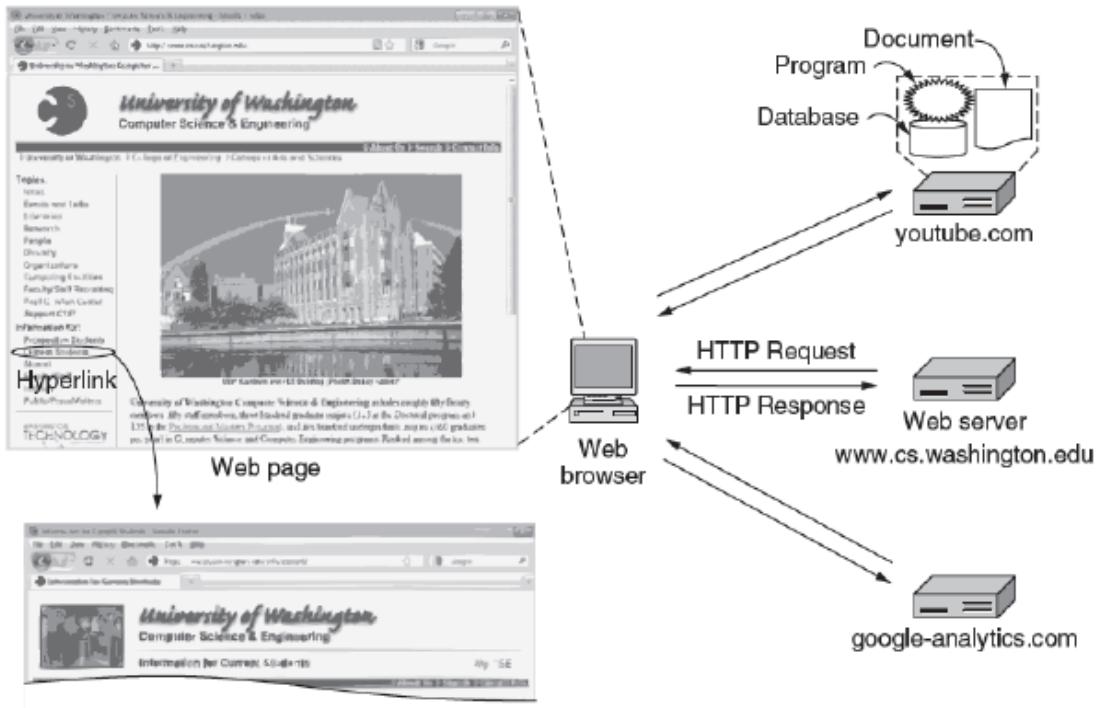
El servidor hace 1 proceso hijo con **fork**, desde el mismo puerto, por cada cliente q quiere conectarse.

## Clase 19: Capa Aplicación - HTTP

El protocolo + utilizado es HTTP. WWW (World Wide Web) es el entorno q utiliza ese protocolo.

Arquitectura:

HTTP transfers pages from servers to browsers



Cada parte de esa página, el servidor, sabe donde ubicarlas utilizando las llamadas:

**URL** (Ubicadores de Recursos Uniformes – Uniform Resource Locators).

: <http://www.phdcomics.com/comics.php>

Protocol              Server              Page on server

Otros protocolos usados:

Our focus →

Name	Used for	Example
http	Hypertext (HTML)	http://www.ee.uwa.edu/~rob/
https	Hypertext with security	https://www.bank.com/accounts/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:///usr/suzanne/prog.c
mailto	Sending email	mailto:JohnUser@acm.org
rtsp	Streaming media	rtsp://youtube.com/montypython.mpg
sip	Multimedia calls	sip:eve@adversary.com
about	Browser information	about:plugins

Procedimiento para q el **cliente** (browser) acceda a la info, luego de cliquear 1 link:

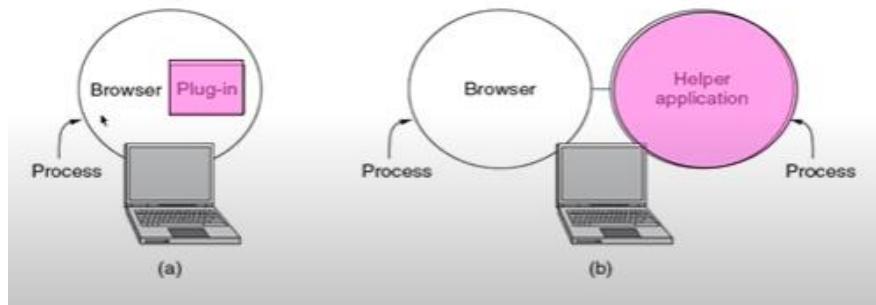
1. Se determina el **protocolo** utilizado (HTTP). (se parsea hasta los :). Además, también se determina el **puerto** a utilizar.
2. Desde el // hasta el 1er / se parsea, y se obtiene el **servidor**. Y se pregunta el DNS de la dirección IP del server.
3. Se hace el saludo de 3 vías. Es decir, se hace la **conexión TCP** con el servidor. Porq www utiliza TCP para intercambiar información, es decir es orientado a conexión en la capa de transporte.
4. Ahora recién, se pide el **requerimiento** por parte del cliente, y el servidor lo responde
5. 1 vez respondido por el servidor, el cliente **lo muestra** por la página.
6. Se cierra idle TCP conexiones, es decir **termina la conexión**.

Pasos del **Servidor**:

1. Se encuentra bloqueada en la llamada de sistema **accept()**, esperando nuevas conexiones.
2. Obtiene el **requerimiento** del cliente y busca si existe el **recurso** solicitado.
3. Si lo obtiene, lo manda al cliente
4. Cierra la conexión.

El cliente sabe cómo interpretar la información y visualizarla gracias al **MIME Types**.

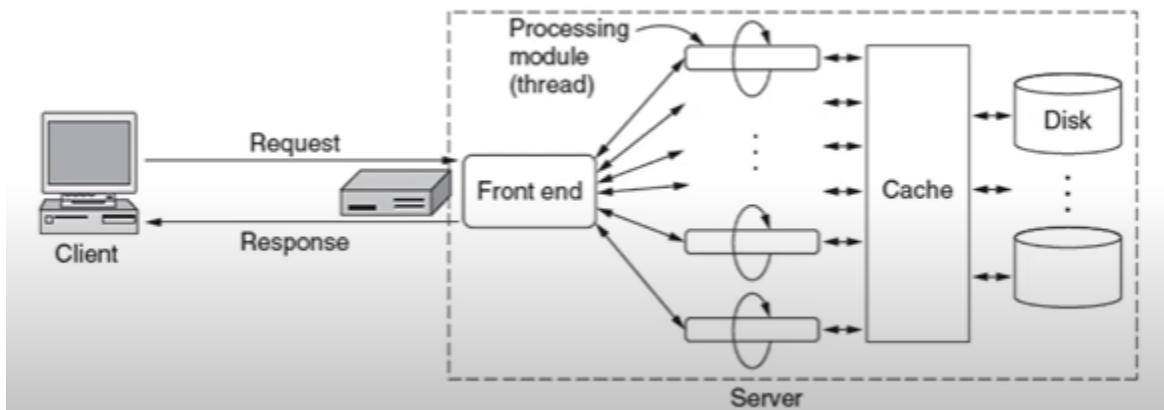
Para los nuevos tipos de datos para representar, lo q se utiliza para incorporarlas a la web son:



(a): Con Plug-In. Para añadirle esa funcionalidad requerida al Browser para poder representar el nuevo tipo de archivo.

(b): Con Helper Application. Se utiliza 1 nuevo proceso para poder representar el nuevo tipo de archivo.

#### Escalar servidores Web:



Se utilizan, **multihilos**, **caches** para almacenar momentáneamente info de discos físicos.

En 1 pagina no existe el concepto de **Sesión**. Es decir, de conectarme a alguna web y hacer todo lo q tenia q hacer, con varias peticiones y después desconectarme.

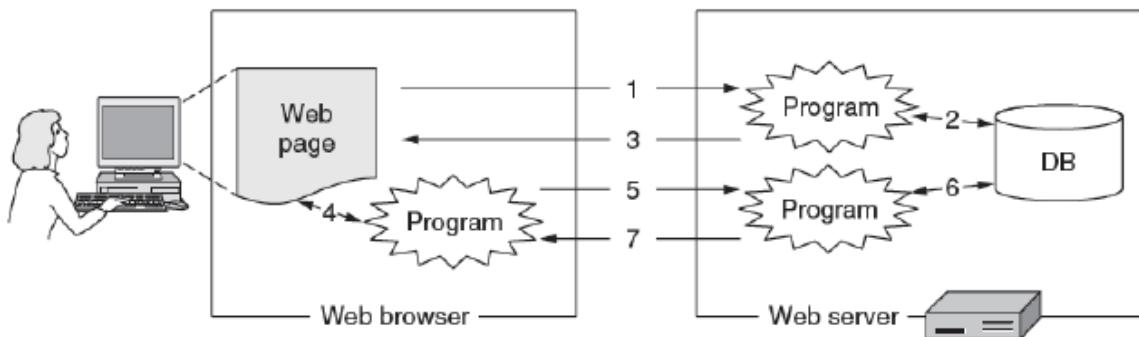
Por lo tanto, la única forma q tiene 1 servidor de saber de que 1 cliente es el mismo q el que se conectó hace 1 rato, es mediante el uso de **Cookies**:

El servidor le envía unos datos para q le cliente q se quiera conectar nuevamente, envíe estos mismos datos, para q el servidor sepa q se trata del mismo cliente.

Domain	Path	Content	Expires	Secure
toms-casino.com	/	CustomerID=297793521	15-10-10 17:00	Yes
jills-store.com	/	Cart=1-00501;1-07031;2-13721	11-1-11 14:22	No
aportal.com	/	Prefs=Stk:CSCO+ORCL;Spt:Jets	31-12-20 23:59	No
sneaky.com	/	UserID=4627239101	31-12-19 23:59	No

### Examples of cookies

### Paginas Dinámicas:



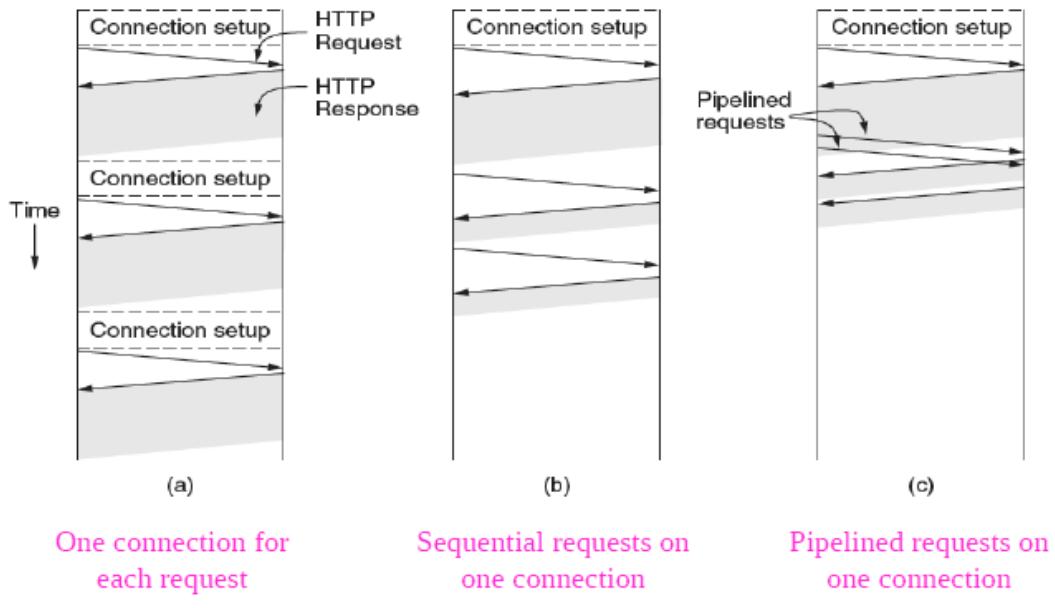
Si existe 1 nueva actualización de la página, el servidor la actualiza, el cliente no tiene q hacer nada desde el navegador.

El servidor corre 1 aplicación o programa, y el resultado de esta aplicación es lo q se manda al cliente. Además también este resultado podría ser 1 nueva aplicación q se corre en el cliente q a su vez esta podría tener nuevos resultados q se envíen al servidor, y así.

## HTTP

Originalmente transfería archivos de hipertexto (HTML). Es 1 protocolo de requerimiento de respuesta. (Yo pido 1 archivo, me lo envían y corta la conexión).

- Del lado del servidor utiliza el puerto 80
- Encabezados en ASCII
- Dependiendo el tipo de archivo, tengo que agregar 1 tipo de dato precediendo al archivo para q el navegador sepa de q tipo de archivo se trata, si debe lanzar 1 plug-in o lanza otra aplicación, etc. Esto es mediante el uso de MIME types.
- Tiene soporte de **pipelining** y de **cache**.
- NO es persistente, se hace el saludo de 3 vias:



(a): para cada requerimiento de archivo se establece 1 nueva conexión.

(b): En 1 actualización de HTTP se puede mantener la conexión establecida, pero periódicamente tengo q estar enviando 1 aviso de q no corte la conexión. Se queda 1 tiempo así, si la dejo de enviar o no hay mas trafico de info se corta la conexión.

(c): Pipeline request. Permite pedir + de 1 cosa a la vez. Primeramente, se pide 1 respuesta, antes de q termine la conexión se requiera otra, y así.

### Métodos de HTTP:

	<b>Method</b>	<b>Description</b>
Fetch a page	→ <b>GET</b>	Read a Web page
	<b>HEAD</b>	Read a Web page's header
Used to send input data to a server program	→ <b>POST</b>	Append to a Web page
	<b>PUT</b>	Store a Web page
	<b>DELETE</b>	Remove the Web page
	<b>TRACE</b>	Echo the incoming request
	<b>CONNECT</b>	Connect through a proxy
	<b>OPTIONS</b>	Query options for a page

### Códigos de respuestas:

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

- Orden de los 100: rango de correos
- Orden de los 200: fue exitoso el pedido
- Orden de los 300: cuando el recurso no esta ahí, esta en otro lado, es decir q me redirecciona hacia donde se encuentra lo solicitado.
- De los 400: error del lado del cliente, se pidió mal la información o no estaba.
- De los 500: error del lado del servidor.

Ejemplos de encabezado:

Function	Example Headers
Browser capabilities (client ↔ server)	User-Agent, Accept, Accept-Charset, Accept-Encoding, Accept-Language
Caching related (mixed directions)	If-Modified-Since, If-None-Match, Date, Last-Modified, Expires, Cache-Control, ETag
Browser context (client ↔ server)	Cookie, Referer, Authorization, Host
Content delivery (server ↔ client)	Content-Encoding, Content-Length, Content-Type, Content-Language, Content-Range, Set-Cookie

Como agilizar el sistema web, es haciendo cache:

