

Resumen Sistemas Operativos y Redes

INDICE

Sistema operativo – Kernel.....	2
Syscalls.....	3
Procesos.....	4
<i>Scheduling</i>	7
<i>Threads</i>	10
Problema de los filósofos.....	13
Memoria.....	14
Segmentación, Paginación, TLB	
Políticas de reemplazo de páginas.....	17
Sistemas de disco.....	19
Asignación de bloques.....	20
Redes.....	22
UDP.....	24
TCP.....	25
Redireccionamiento de red.....	26

Sistema operativo

El sistema operativo provee una interfaz de llamadas al sistema (*syscall*)

- Cada vez que un programa desea solicitar algo al sistema operativo, debe invocar una *syscall*.

El sistema operativo provee interfaces de usuario (*user interface*) que enmascara *syscalls*

- Ofrecen un entorno más “amigable” para usar el computador.

Distintas maneras de interactuar con el sistema operativo

- Interfaces gráficas: **GUI** (*Graphical User Interface*).
- Intérprete de comandos (**Command Line**).
- **Batch** (lotes): Secuencias de comandos no-interactivos.

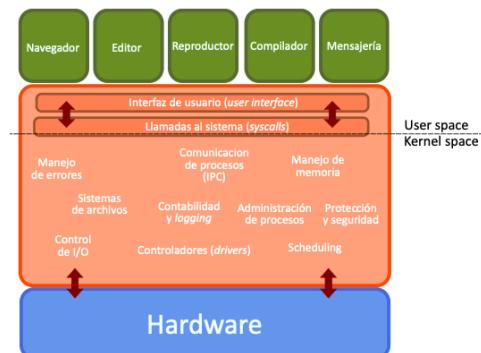
El **kernel** (núcleo, supervisor)

- El código fundamental del sistema operativo.
- Tiene completo control sobre el sistema (*hardware*).
- Sistema operativo incluye herramientas (programas) que facilitan la labor del *kernel*, pero que no son fundamentales.

Kernel monolítico

Todo dentro de un único programa

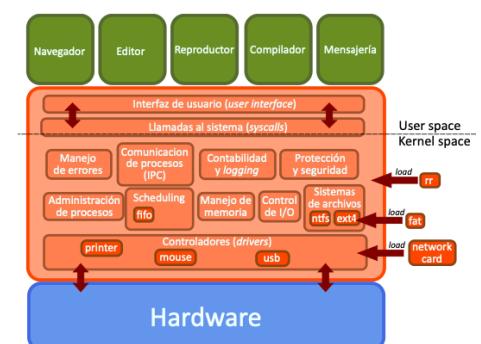
- Servicios para el sistema y para el usuario en el mismo espacio.
- Todos los servicios se ejecutan en modo *kernel*.
- La falla de un servicio compromete al *kernel*.
- *Kernel* complejo y de mayor tamaño
- Ejecución más rápida.



Kernel monolítico con módulos

Todo en un único programa extensible.

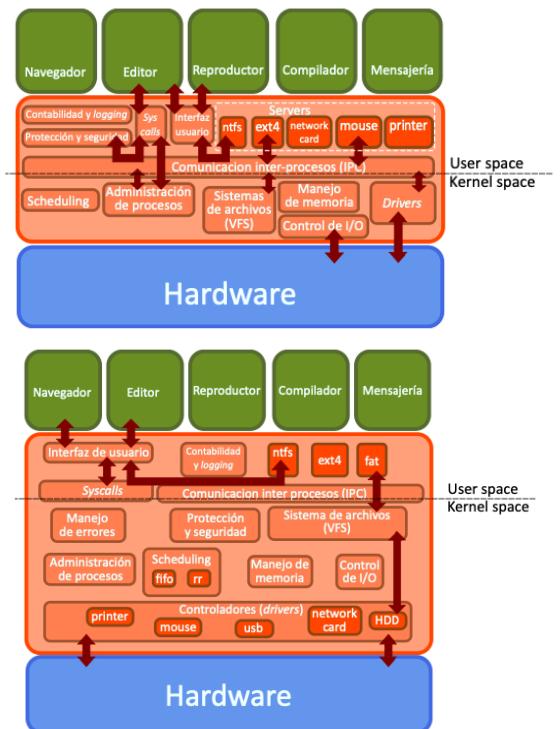
- Componentes pueden ser incluidos al momento de compilar, o bien durante la ejecución como módulos.
- Módulos extienden al *kernel*.
- Módulos se ejecutan en el espacio del *kernel*.
- Ayuda a controlar el tamaño del *kernel* (*footprint*).



Microkernel

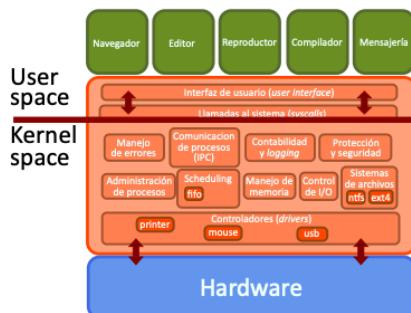
Mantener el *kernel* del menor tamaño posible

- Solo servicios básicos en el *kernel*.
 - Funcionalidades (*servers*) se ejecutan en *user space*.
 - Errores en los servicios no afectan al *kernel*.
 - Procesos se comunican (IPC) mediante paso de mensajes.



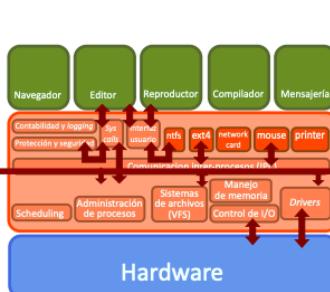
Monolítico

- *Kernel* de mayor tamaño y complejo
 - Todos los servicios en *kernel space*
 - Falla en un servicio afecta a todo el *kernel*
 - Ejecución más rápida
 - Difícil de extender, pero puede usar módulos



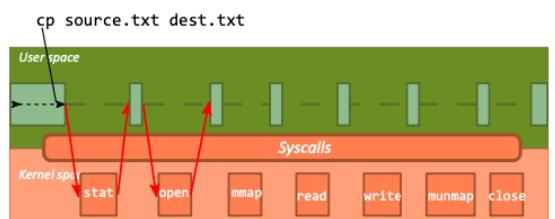
Microkernel

- Kernel pequeño y simple
 - Servicios mínimos en *kernel space*. Mayoría en *user space*
 - Falla en un servicio queda aislada
 - Ejecución con *overhead* por comunicación
 - Fácil de extender



Syscalls: Llamadas al sistema

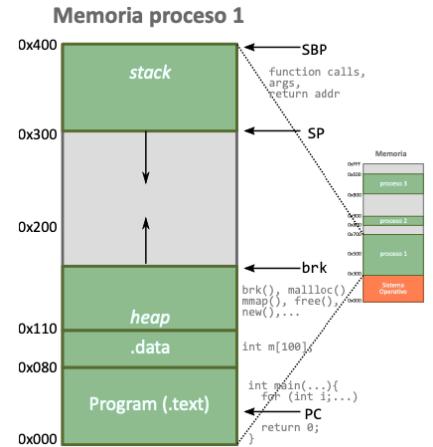
- Abre source.txt
 - Abre dest.txt
 - Leer source.txt hacia alguna región de la memoria (mmap, read)
 - Escribir desde la memoria hacia dest.txt (write)
 - Liberar la memoria y guardar los archivos (munmap, close).



Procesos

¿Qué hay en un proceso?

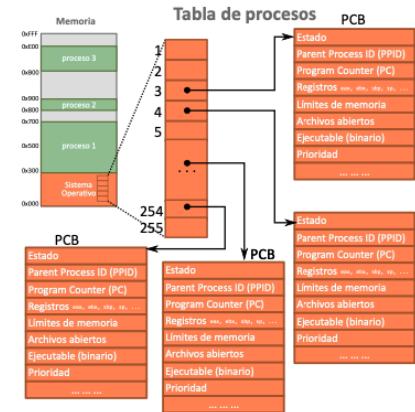
- **Código** (.text, información estática).
- **Datos** (.data): Variables globales.
- **Stack**: Cada ítem del stack representa un llamado a función (*call frame*), y contiene:
 - Parámetros
 - Variables locales
 - Lugar de retorno (donde estaba la ejecución anterior, PC)
- **Heap**: Memoria asignada dinámicamente (durante la ejecución).



Process Control Block (PCB)

Sistema operativo mantiene una **tabla de procesos**. Información de cada proceso almacenada en su PCB.

- Estado.
- Identificador (PID).
- Program counter (PC).
- Registros de CPU.



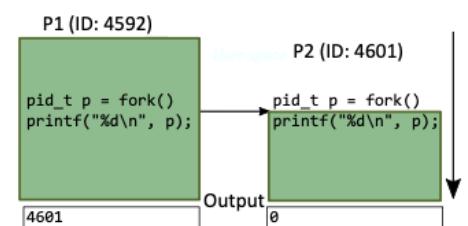
Un proceso en ejecución puede cambiar de estado.

- **New**: En creación
- **Running**: En ejecución
- **Waiting**: Esperando (I/O signal)
- **Ready**: Listo para ejecutar. Esperando asignación de CPU
- **Terminated**: Ejecución terminada/



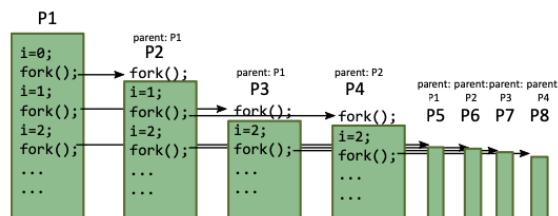
Syscall fork()

- Crea un nuevo proceso como **copia** del padre.
- Ambos continúan ejecutando desde la instrucción de retorno de `fork()`
- `fork()` retorna PID del hijo al padre, y retorna 0 al hijo.



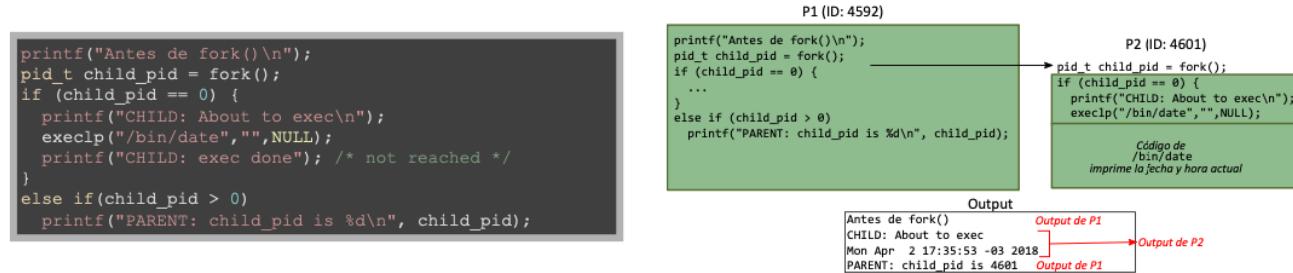
```

/* ... */
for(int i=0; i<4; i++) {
    fork();
    printf("[%d] %d\n", getpid(), i);
}
  
```



Syscall exec()

- Carga un binario en memoria **reemplazando** el código de quien llamó, e inicia su ejecución.
- El programa nuevo se “roba” el proceso (la memoria se sobrescribe).



- El proceso PID 4601 se convierte en /bin/date.
- La línea CHILD: exec done nunca se imprime (salvo que exec falle).
- exec no retorna (salvo si falla, retorna -1). Falla cuando no puede lanzar el proceso.
- El proceso PID 4601 sigue siendo hijo de PID 4592, pero ahora su código y memoria son distintos.
- Tanto PID 4592 como PID 4601 siguen ejecutando concurrentemente.

Syscall wait()

- El proceso padre espera a que uno o más de sus procesos hijos finalicen su ejecución.
- Suspende la ejecución del proceso padre hasta que al menos uno de sus procesos hijos haya terminado.
- Cuando uno de los hijos termina, el *syscall* *wait()* devuelve el ID del proceso hijo que finalizó.



- El proceso PID 4592 se bloquea (estado *waiting*) hasta que su hijo termina.
- Cuando el hijo termina, el sistema operativo muda a PID 4592 de regreso a estado *ready*.
- El proceso PID 4601 sigue siendo hijo de PID 4592, por lo tanto, PID 4601 sabe cuándo el hijo ha terminado.

Syscall exit()

- Se utiliza para terminar la ejecución de un proceso.
- Cuando se llama a exit(), el proceso actual se cierra inmediatamente y el control vuelve al sistema operativo.



La syscall `wait()` permite recibir un valor del hijo, desde donde se puede extraer el `exit code`.

Syscall kill()

- Envía una **señal** a otro proceso (por defecto SIGTERM).
- `kill -l` permite ver las señales disponibles.
- SIGTERM indica al proceso que debe terminar.
- SIGKILL elimina al proceso de la tabla de procesos (sin piedad).
- KILLALL permite enviar una **señal** a un grupo de procesos.

Syscall sleep()

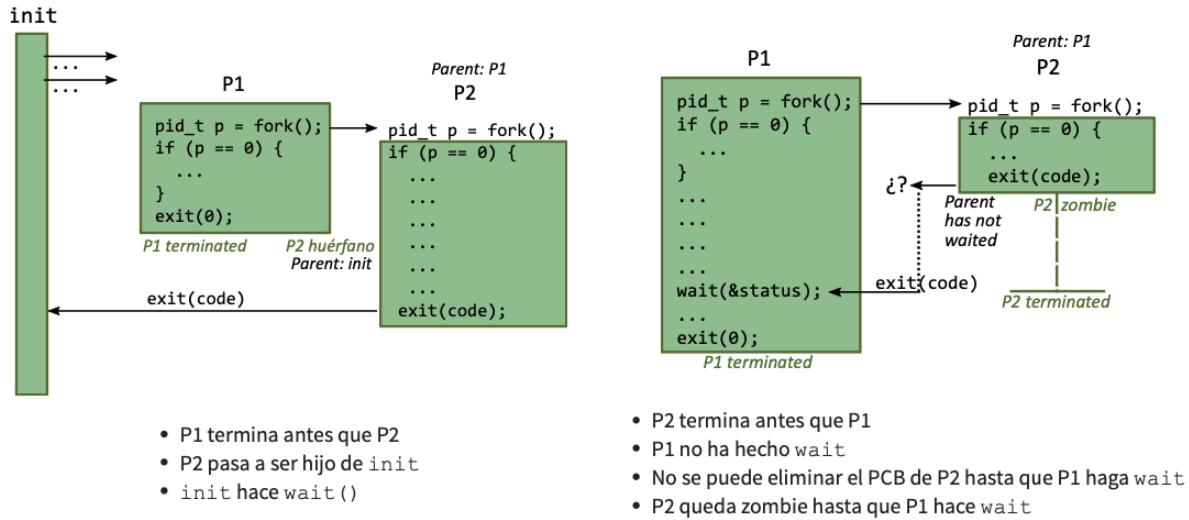
- Se utiliza para suspender la ejecución de un proceso durante un tiempo determinado, lo que provoca que el proceso se “duerma” o entre en un estado de espera.
- Durante el tiempo especificado, el proceso no realizará ninguna acción y cederá su tiempo de CPU a otros procesos que estén listos para ejecutarse.

Procesos huérfanos

- Cuando un padre termina (`exit`) o muere (`kill`), sus hijos quedan huérfanos y pasan a ser hijos de `init`.
- Sin embargo, podría parecer que los hijos sí mueren (las condiciones pueden ser complicadas).
- `init` hace `wait()` periódicamente por sus hijos.

Procesos zombies

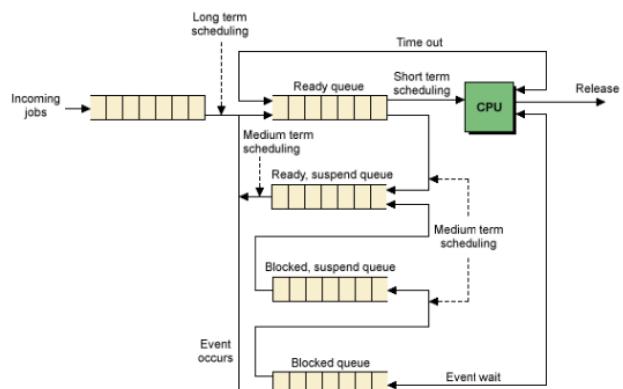
- Cuando un proceso termina y su padre no hace `wait()`.
- Proceso terminado no se borra inmediatamente de la tabla de procesos (pero tampoco ejecuta).
- Proceso queda en estado *zombie* hasta que el padre hace `wait()`.



Scheduling

Proceso de decidir qué proceso o tarea se ejecutará a continuación en una unidad de procesamiento central (CPU).

- **Long-term Scheduler**
 - Admite procesos en la *cola ready*.
 - Determina el **grado de multiprogramación** (cantidad de procesos en memoria).
- **Short-term Scheduler (a.k.a. dispatcher)**
 - Selecciona un proceso de la *cola ready* para ejecutar.
 - Ejecuta el cambio de contexto
- **Medium-term Scheduler**
 - Modifica temporalmente el grado de multiprogramación
 - Ejecuta **swapping** copiando memoria RAM a disco, y de disco a RAM.



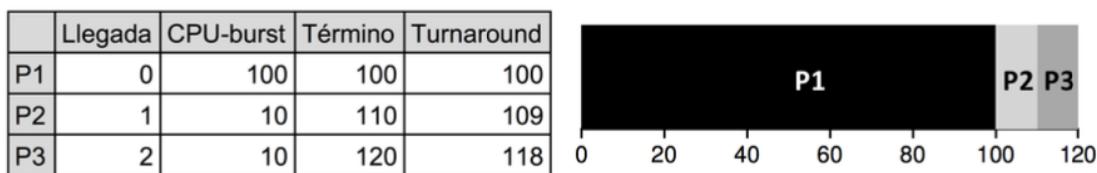
BATCH SCHEDULING

Trabajo por lotes. Sin interacción.

- Mantener la CPU lo más ocupada posible.
- Minimizar **turnaround time**: tiempo desde envío a término
- Maximizar **throughput**: número de trabajos por hora.

- **FIRST-COME, FIRST-SERVED (FCFS)**

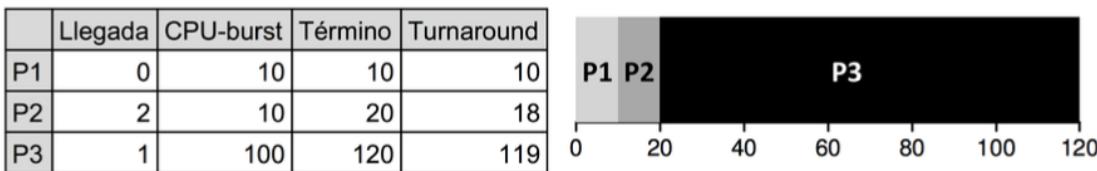
Orden de llegada. Cola FIFO.



Turnaround time promedio: 109

Si P2 hubiese llegado en $t = 0$, y P1 hubiese llegado en $t = 1$, entonces
turnaround time promedio $\rightarrow 79$

- **SHORTEST JOB FIRST (JSF)**



Turnaround time promedio: 49

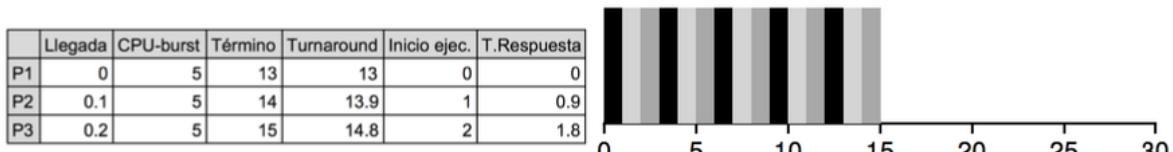
INTERACTIVE

Métrica: Tiempo de respuesta (*response time*).

Tiempo desde llegada (a la cola) hasta la primera ejecución.

- **ROUND-ROBIN (RR)**

Un turno para cada uno. Ejemplo con $q = 1$.



- Turnaround time promedio: 13.9
- Response time promedio: 0.9

- **PRIORITY SCHEDULING**

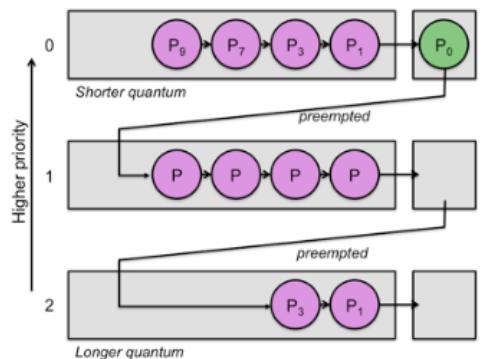
Cada proceso tiene una **prioridad** asociada.

- Se atienden por orden de **prioridad**.
- Prioridades iguales: FCFS o RR.
- Prioridades pueden ser estáticas o dinámicas.

- **MULTILEVEL FEEDBACK QUEUE (MLFQ)**

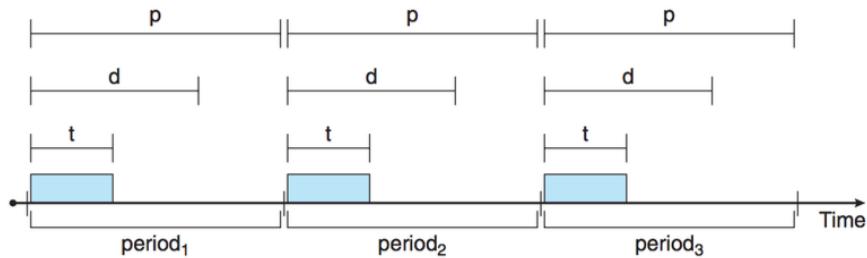
Múltiples colas con distinta prioridad.

- R1. Si $\text{priority}(A) > \text{priority}(B)$, ejecutar A .
- R2. Si $\text{priority}(A) = \text{priority}(B)$, ejecutar A y B con RR.
- R3. Procesos entran en la cola con **mayor** prioridad.
- R4. Si un proceso usa su q (acumulado en todos sus turnos), su prioridad se reduce.
- R5. Después de un tiempo S , todos los procesos se mueven a la cola con mayor prioridad.



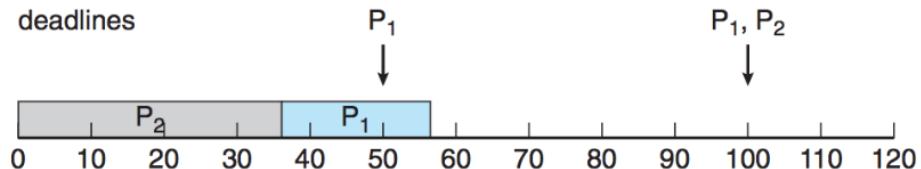
REAL TIME

- Poseen *deadlines* y periodos de ejecución.
- Sistema debe determinar si, dado *deadline* (d), periodo (p) y tiempo de ejecución (t), es capaz de incorporar el proceso para la ejecución.

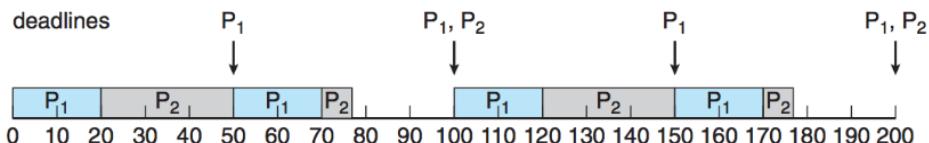


- **RATE MONOTONIC SCHEDULING (RMS)**

- Cada proceso se asocia con un período y una prioridad basada en su período. Cuanto más corto es el período de un proceso, mayor es su prioridad.
- La idea central del algoritmo es que los procesos con plazos más cortos (mayor frecuencia) son más críticos y deben tener prioridad para ser ejecutados antes que los procesos con plazos más largos.
- Tiene prioridades estáticas.
- Tiempo de ejecución determinista.



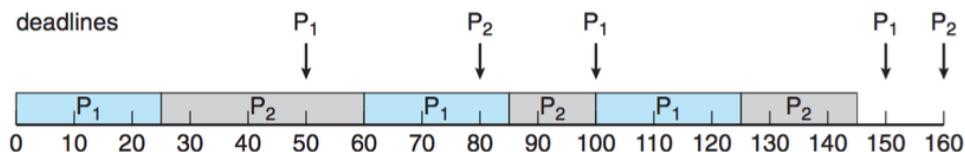
Cada proceso recibe prioridad: $\frac{t_i}{p_i}$
Con P₁ : {p₁ = 50, t₁ = 20} , y P₂ : {p₂ = 100, t₂ = 35}



- **EARLIEST DEADLINE FIRST SCHEDULING (EDF)**

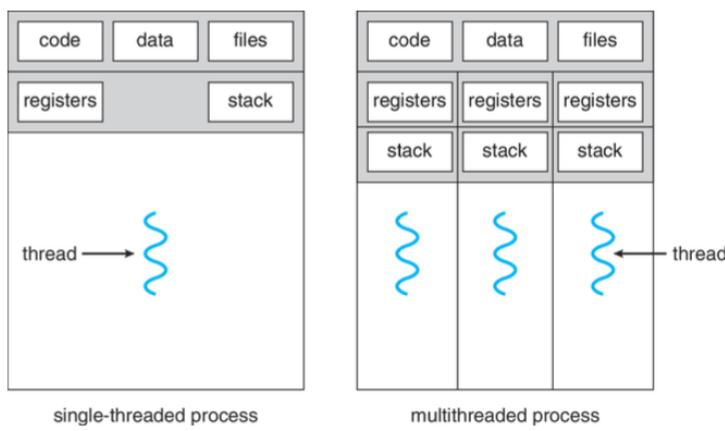
- Asigna prioridades dinámicamente a los procesos en función de sus plazos (*deadlines*) más cercanos.
- El proceso con el deadline más cercano tiene la máxima prioridad y se planifica para ejecutarse primero.
- Prioridades cambian dinámicamente en tiempo de ejecución. En cada momento, el proceso con el deadline más cercano se selecciona y tiene la máxima prioridad.
- Tiempo de ejecución determinista.

Con P₁ : {p₁ = 50, t₁ = 25} , y P₂ : {p₂ = 80, t₂ = 35}



Threads

Es como un proceso, pero más liviano. Procesos pueden tener uno o más procesos.



Los *threads* comparten:

- Espacio de memoria.
- Variables globales.
- Archivos abiertos.
- Procesos hijo.
- Señales y manejadores de señales.
- Información de *accountability*.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Los *threads* NO comparten:

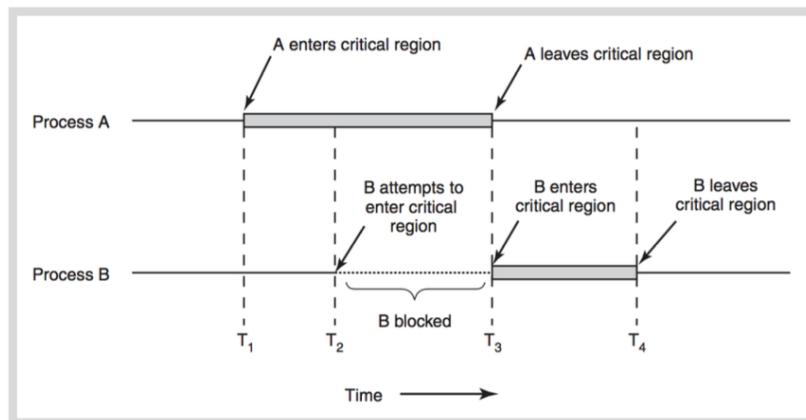
- *Stack*.
- Registros (incluyendo PC).
- Estado.

Race Condition

Situación en que la salida de una operación depende del orden temporal de sus operaciones internas, el cual no está bajo control del programador.

Sección crítica

- El segmento de código en que el *thread* accede a recursos compartidos se conoce como sección crítica.
- Necesitamos un protocolo que **no** permita que dos o más *threads* se encuentren en su sección crítica el mismo tiempo. Para esto se debe cumplir:
 - Exclusión mutua: A lo más un *thread* está en su SC.
 - Progreso: Al menos un *thread* puede entrar a su SC. Si ningún *thread* está en su SC, y hay *threads* que desean entrar, entonces los que quieren entrar deben decidir quién entra, y decidirlo en un tiempo acotado.
 - Espera acotada: Si un proceso quiere entrar a su SC, podrá hacerlo luego de una cantidad **finita** de tiempo.



MUTEX LOCKS

Lo más simple: *MUTual EXclusion locks* (mutex locks)

- `Lock::acquire();` toma el *lock*
- `Lock::reléase();` libera el *lock* (error si no está tomado)

```
struct Lock
{
    bool value = false; // false==free, true==busy
};

void acquire()
{
    while (test_and_set(&value))
    {
        thread_yield();
    }
}

void release()
{
    value = false;
};
```

Solución a la sección crítica

```
while (true)
{
    /* ... */
    lock.acquire();
    /* ... SC ... */
    lock.release();

    /* ... Out of SC ... */
}
```

Contador con acceso exclusivo

```
struct Lock lock; // compartido

for (int i = 0; i < max; i++)
{
    lock.acquire();
    counter = counter + 1;
    lock.release();
};
```

SEMÁFOROS

- Permiten un número limitado de *threads* en una sección crítica.
- Un semáforo *S* incluye un contador y dos operaciones:
 - *proberen P()* o *wait()* o *down()*. Intenta decrementar el valor.
 - *verhogen V()* o *signal()* o *up()*. Incrementa el valor.

```
struct semaphore
{
    int count;
    struct Lock l;
    struct process *slept = NULL;
};

void init(c)
{
    count = c;
}

void P()
{
    /* wait: put into queue and sleep */
    l.acquire();
    while (count <= 0)
    {
        /* code to add itself to 'slept' */ //
        l.release();
        sleep();
        l.acquire();
    }
    count--;
    l.release();
}

void V(S)
{
    /* signal */
    l.acquire();
    count++;
    wakeup(/* first from 'slept' */);
    // plus, remove from queue
    l.release();
}
```

Problema de los filósofos

```

#define NUM_HILOSOFOS 5

pthread_mutex_t tenedores[NUM_HILOSOFOS];
pthread_t filosofos[NUM_HILOSOFOS];

void *filosofo(void *arg) {
    int id = *(int *)arg;
    int tenedor_izq = id;
    int tenedor_der = (id + 1) % NUM_HILOSOFOS;

    while (1) {
        // Filósofo piensa
        printf("Filósofo %d está pensando ... \n", id);
        sleep(2);

        // Filósofo quiere comer
        printf("Filósofo %d quiere comer ... \n", id);

        // Intenta tomar el tenedor izquierdo
        pthread_mutex_lock(&tenedores[tenedor_izq]);
        printf("Filósofo %d ha tomado el tenedor izquierdo (%d). \n", id, tenedor_izq);

        // Intenta tomar el tenedor derecho
        if (pthread_mutex_trylock(&tenedores[tenedor_der]) == 0) {
            printf("Filósofo %d ha tomado el tenedor derecho (%d). \n", id, tenedor_der);

            // Filósofo come
            printf("Filósofo %d está comiendo ... \n", id);
            sleep(3);

            // Libera los tenedores
            pthread_mutex_unlock(&tenedores[tenedor_izq]);
            pthread_mutex_unlock(&tenedores[tenedor_der]);

            printf("Filósofo %d ha terminado de comer y ha soltado ambos tenedores. \n", id);
        } else {
            // Si no puede tomar el tenedor derecho, suelta el izquierdo y vuelve a pensar
            pthread_mutex_unlock(&tenedores[tenedor_izq]);
            printf("Filósofo %d no pudo tomar el tenedor derecho, ha soltado el tenedor izquierdo y seguirá pensando ... \n", id);
        }
    }
    return NULL;
}

int main() {
    int i;
    int fil_ids[NUM_HILOSOFOS];

    // Inicializa los tenedores
    for (i = 0; i < NUM_HILOSOFOS; i++) {
        pthread_mutex_init(&tenedores[i], NULL);
    }

    // Crea los filósofos
    for (i = 0; i < NUM_HILOSOFOS; i++) {
        fil_ids[i] = i;
        pthread_create(&filosofos[i], NULL, filosofo, &fil_ids[i]);
    }

    // Espera a que los filósofos terminen (esto no sucederá nunca debido al ciclo infinito)
    for (i = 0; i < NUM_HILOSOFOS; i++) {
        pthread_join(filosofos[i], NULL);
    }

    // Libera los tenedores
    for (i = 0; i < NUM_HILOSOFOS; i++) {
        pthread_mutex_destroy(&tenedores[i]);
    }

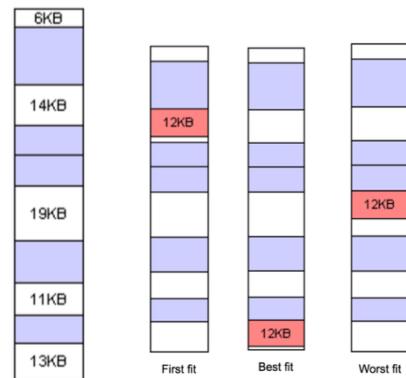
    return 0;
}

```

Memoria

¿Cómo evitar espacios libres?

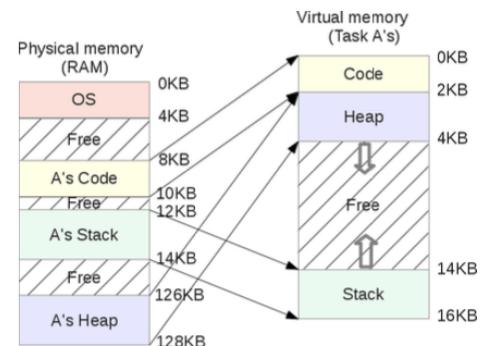
- **First-Fit.** En el primer lugar disponible.
- **Best-Fit.** En el que deja menos espacio libre.
- **Worst-Fit.** En el que deja más espacio libre.



Con cualquier estrategia aún pueden quedar espacios libres no contiguos.

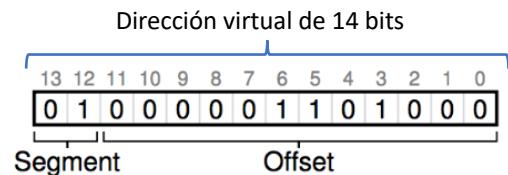
Segmentación

- Se divide el espacio de direcciones en varios espacios más pequeños.
- Espacios más pequeños pueden ser asignados más fácilmente.
- La **MMU** debe ser capaz de hacer la traducción de memoria física a virtual.



¿Cómo conocer segmento y offset?

Si tenemos 3 segmentos, podemos usar 2 bits para identificarlos



El **offset** es la parte de una dirección de memoria que representa la posición o desplazamiento dentro del segmento. En otras palabras, indica la distancia desde el inicio del segmento hasta la ubicación específica de un dato o instrucción.

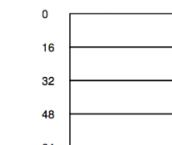
Paginación

Problema: Un espacio de direcciones muy grande.

- Idea 1: Dividir el espacio de direcciones en **segmentos**.
- Idea 2: Que los segmentos sean del mismo tamaño: **páginas de memoria**.

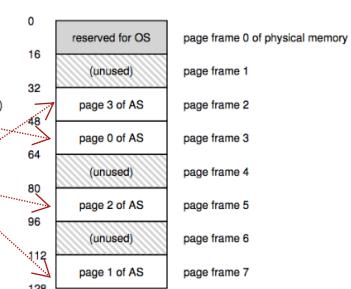
- Espacio virtual: **páginas**
- Espacio físico: **frames** (marcos)
- Páginas y frames del **mismo** tamaño

Espacio virtual 64 B



Dividido en 4 páginas,
cada una de 16 B

Espacio físico 128 B



8 frames de espacio
físico. Cada uno de 16 B

Páginas de 16B, espacio virtual de 64B, espacio físico de 128B.

- Direcciones posibles en una página: $16 = 2^4$ Byte \rightarrow 4 bit.
 - 4 bits indican el *offset* dentro de una página.
- Espacio virtual de $64 = 2^6$ Byte. Requiere 6 Bits.
 - 2 bits indican el número de página.
 - Páginas posibles: $2^2 = 4$ entradas en la tabla de páginas.
- Espacio físico de $128 = 2^7$ Byte. Requiere 7 Bits.
 - 3 bits indican el número de *frames* (PFN).
 - *Frames* en la memoria principal: $2^3 = 8$ *frames*.

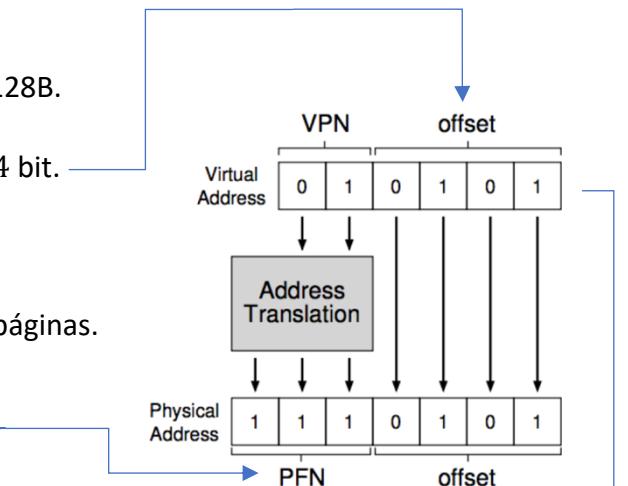


Tabla de páginas de ejemplo:

Page (VPN)	Frame (PFN)
0 (0b00)	3 (0b011)
1 (0b01)	7 (0b111)
2 (0b10)	5 (0b101)
3 (0b11)	2 (0b010)

Dirección virtual: 21 (0b010101)

- # Página: 1 (0b01)
- Offset: 5 (0b0101)
- Tabla: 1 → 7

Dirección física: 117 (0b1110101)

Otro ejemplo:

Páginas de 4KB, dirección virtual de 16 Bits, dirección física de 15 Bits.

$$2^{10} B = 1024 B = 1 KB$$

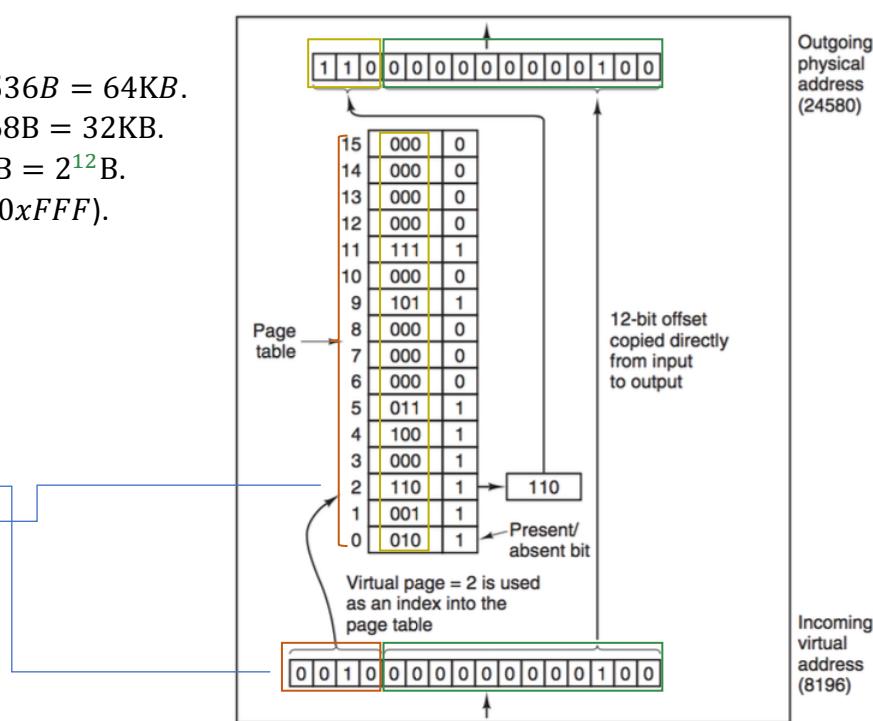
Dimensiones:

- Tamaño espacio virtual: $2^{16} B = 65536 B = 64 KB$.
- Tamaño espacio físico: $2^{15} B = 32768 B = 32 KB$.
- Tamaño de página: 4KB = $2^2 \times 2^{10} B = 2^{12} B$.
- Bits para offset: 12 (rango: 0x000 a 0xFFFF).
- Bits para #página: 4 (16 páginas).
- Bits para #frame: 3 (8 frames).

Traducción

- Dirección lógica: 8196 (0x2004).
 - #página: 2
 - #frame: 6
- Dirección física: 24580 (0x6004).

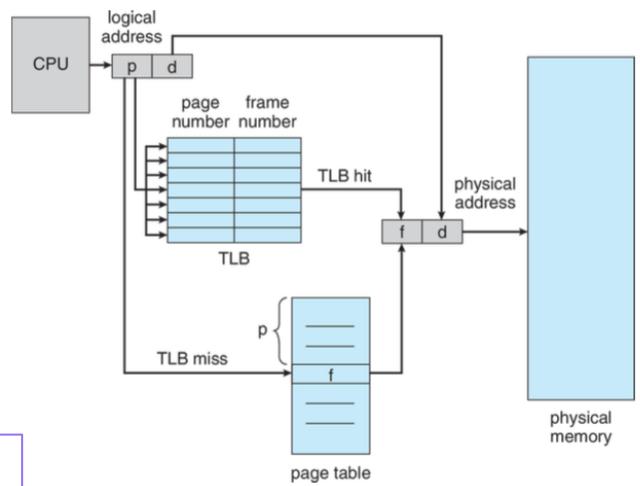
Tamaño de página:
 $16 \times 4 = 64$ Bit = 8 B



Paginación con Translation Look-aside Buffer (TLB)

- Si la dirección está en el caché (TLB hit), se responde directamente.
- Si la dirección no está en el caché (TLB miss), se lee desde la memoria y se actualiza la TLB.

```
int sum = 0;
for (int i = 0; i < 10; i++) {
    sum += a[i];
}
```



Ejemplo: espacio virtual de 8 bit, con páginas de 16 B, almacenando arreglo de 10 int (4b).

- 4 bits para offset. 4 bits para #página. 16 páginas.
- Considerando sólo el acceso a "a"
- Sin TLB: $10 \times (\text{acceso tabla de páginas, acceso a } a[i])$
 - 20 accesos a memoria.
- Con TLB: *miss, hit, hit, miss, hit, hit, hit, miss, hit, hit*.
 - 10 accesos a $a[i]$, 3 accesos a tabla de página (*miss*).
 - 13 accesos a memoria.
 - 7 de cada 10 accesos ahorrados: 70% TLB Hit Rate

	Offset	00	04	08	12	16
VPN = 00						
VPN = 01						
VPN = 02						
VPN = 03						
VPN = 04						
VPN = 05						
VPN = 06		a[0]	a[1]	a[2]		
VPN = 07		a[3]	a[4]	a[5]	a[6]	
VPN = 08		a[7]	a[8]	a[9]		
VPN = 09						
VPN = 10						
VPN = 11						
VPN = 12						
VPN = 13						

VF 1 miss por cada vez que se
VF accede por primera vez a una fila

Localidad espacial

Después de acceder a una dirección x, el programa probablemente accederá a direcciones cercanas a x.

- Luego de un *miss*, y guardar un PTE en TLB, vendrán muchos *hit*.

Localidad temporal

Una dirección x que acaba de ser accedida, probablemente será accedida de nuevo dentro de poco tiempo.

- Una PTE recién guardada, volverá a ser usada pronto.

Reemplazo de páginas

SWAP SPACE

Páginas a ser borradas de memoria van a espacio de swap.

- Representan parte del espacio de memoria de un proceso en ejecución.
- Tabla de páginas utiliza *present bit* para saber si la página está en un *frame* o no

Si la página no está presente, se genera un *page fault*

Page fault activa el mecanismo de recuperación de una página desde disco a memoria física.

- Sistema operativo atiende el *page fault*.
- Cuando el *page fault* ha sido resuelto, el proceso puede continuar.

```
VPN = (virtualAddress & VPN_MASK) >> VPN_SHIFT;
(Success, TLBEntry) = TLB_Lookup(VPN);
if (Success)
{ // TLB Hit
    if (!TLBEntry.protected())
    {
        offset = virtualAddress & OFFSET_MASK;
        physicalAddress = (TLBEntry.PFN << PFN_SHIFT) | offset;
        register = ReadMemory(physicalAddress);
    }
    else
        raise(PROTECTION_FAULT);
}
else
{ // TLB Miss
    PTEAddress = PageTableBaseRegister + (VPN * sizeof(PTE));
    PTE = ReadMemory(PTEAddress); // lectura tabla de páginas
    if (!PTE.valid)
        raise(SEGFAULT);
    else if (PTE.protected)
        raise(PROTECTION_FAULT);
    else if (!PTE.present)
        raise(PAGE_FAULT);
    else
    { // pagina en memoria
        TLB_Insert(VPN, PTE);
        Retry();
    }
}
```

Algoritmo para paginación con TLB, y *page faults*

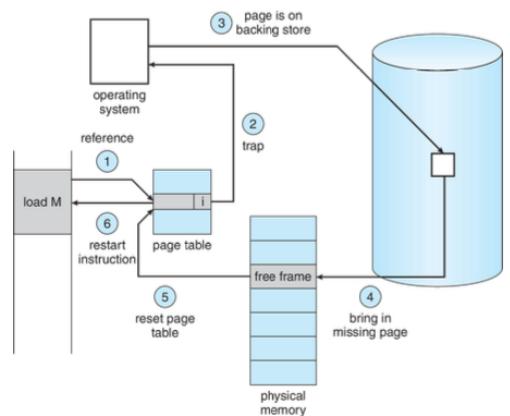
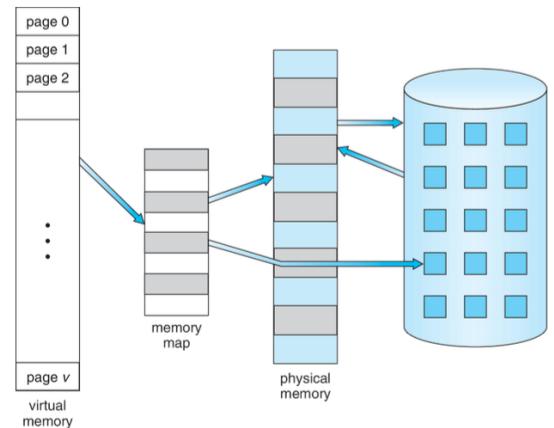
```
PFN = FindFreePhysicalPage(); // busca frame libre
if (PFN == -1) // no había frame libre :(
    PFN = ReplacePage(); // rutina de reemplazo de página

// Copia desde el disco a la memoria. Proceso queda "waiting on I/O"
DiskRead(PTE.DiskAddress, PFN);

// Actualiza tabla de páginas
PTE.present = true;
PTE.PFN = PFN;

// Vuelve a ejecutar la instrucción que generó el PAGE_FAULT
Retry();
```

Algoritmo para manejar un *page fault*



La ocurrencia de un page-fault es algo malo (significa acceder al disco)

Cualquier algoritmo de decisión debe intentar minimizar la ocurrencia de *page faults*

MIN: REEMPLAZO ÓPTIMO

- Se elige la página que será usada lo más lejos posible en el futuro para reemplazarla.
- Es el mejor algoritmo.
- Pero no podemos predecir el futuro.

Ejemplo: Sistema con 3 frames como máximo para un proceso.

Secuencia de accesos a páginas: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1

Página	Hit/Miss	Reemplazar	Páginas en mem.
0	Miss	0	
1	Miss	0, 1	
2	Miss	0, 1, 2	
0	Hit	0, 1, 2	
1	Hit	0, 1, 2	
3	Miss	2	0, 1, 3
0	Hit	0, 1, 3	
3	Hit	0, 1, 3	
1	Hit	0, 1, 3	
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Tasa de hits (*Hit Rate*)

$$\text{Hit Rate} = \frac{6}{6+5} = 0.545$$

Si ignoramos el primer miss de cada página
(*cold start*)

$$\text{Hit Rate} = \frac{6}{6+2} = 0.75$$

FIFO: FIRST IN FIRST OUT

- Elige la página que lleva más tiempo en memoria.

Página	Hit/Miss	Reemplazar	Páginas en mem.
0	Miss	0	
1	Miss	0, 1	
2	Miss	0, 1, 2	
0	Hit	0, 1, 2	
1	Hit	0, 1, 2	
3	Miss	2	0, 1, 3
0	Hit	0, 1, 3	
3	Hit	0, 1, 3	
1	Hit	0, 1, 3	
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Tasa de hits (*Hit Rate*)

$$\text{Hit Rate} = \frac{6}{6+5} = 0.545$$

Si ignoramos el primer miss de cada página
(*cold start*)

$$\text{Hit Rate} = \frac{6}{6+2} = 0.75$$

RANDOM

- Elige una página aleatoria.
- Podría eliminar páginas que van a usarse en el corto plazo.

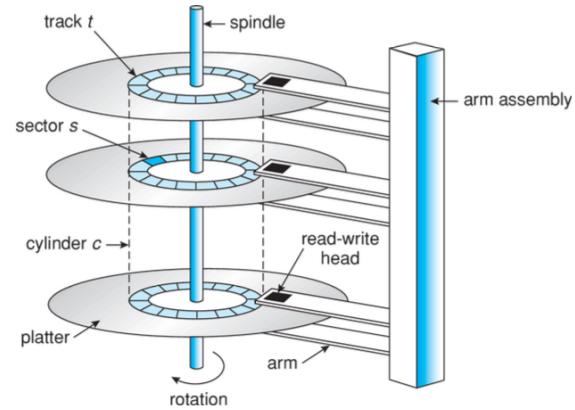
LRU: LAST RECENTLY USED

- Elige la página que lleva más tiempo sin usarse.

SISTEMAS DE DISCO

Discos magnéticos

- **Brazos (arms)** se mueven juntos.
- **Brazos** poseen cabezas lectoras (**heads**).
- **Platos (platter)** divididos en **tracks** circulares.
- **Tracks** divididos en **sectores**: bloques de 512 bytes.
- Conjunto de **tracks** entre varios platos forman un cilindro



ARCHIVOS

Archivos

- Creación: open
- Lectura/Escritura: read, write
- Lectura/Escritura (no secuencial): lseek
- Escritura inmediata: fsync
- Renaming: mv
- Estado: stat
- Eliminación: rm

Directorios

- Creación: mkdir
- Lectura: opendir, readdir, closedir
- Eliminación: rmdir

Montando sistemas de archivos

Sistema puede manejar múltiples **sistema de archivos** en discos (dispositivos de bloques)

Para acceder a sistemas de archivos se necesita un punto de acceso, o *mountpoint*.

Sistema operativo integra el sistema de archivos a la jerarquía actual usando *mount*.

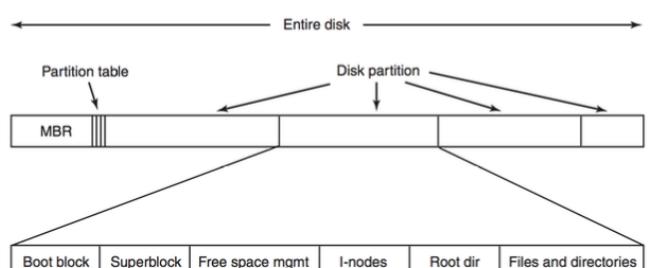
```
mount /dev/sda1 /home/users
```



El **sistema de archivos** permite obtener una dirección de disco (un bloque) a partir de un conjunto de datos simbólicos: nombre de archivo, directorio, rutas, links, ...

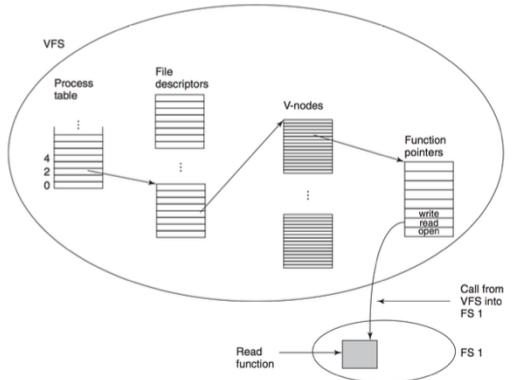
Master Boot Record (MBR). Sector 0. Indica *Partition Table*

- **Boot Control Block**. Uno por cada partición. Permite cargar el sistema operativo.
 - Unix: Boot Block
 - Windows: Partition Boot Sector
- **Volume Control Block**. Datos de formato del sistema de archivos.
 - Tipo de sistema de archivos, números de bloques,...
 - Unix: Superblock
 - Windows: Master File Table.
- **Estructura de directorio**. Archivos.



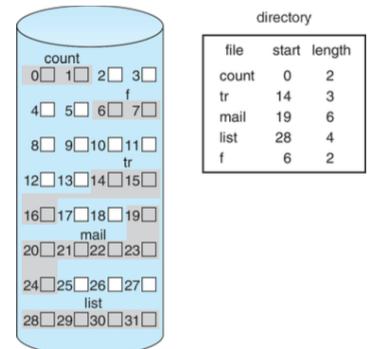
Virtual File System (VFS)

- **Tabla de archivo abiertos** mantiene *file descriptors*. Estructuras en memoria con información de archivo.
- **v-node** representa el punto de acceso al archivo. Cada **v-node** apunta a funciones del sistema de archivos que ejecutan las llamadas concretas.



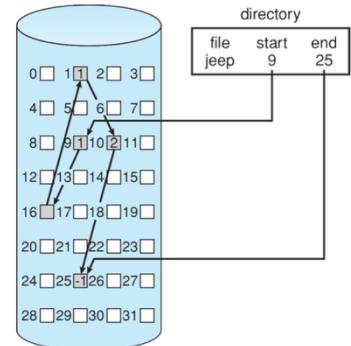
Asignación contigua

- Archivo de n bloques: $b, b + 1, \dots, b + n - 1$.
- Fácil de acceder al siguiente bloque.
- Problema: Fragmentación externa. Si quiero crear un archivo que ocupe 7 bloques, no me dejará porque no hay 7 bloques contiguos disponibles. Pero si hay más de 7 en total.
- Requiere operaciones de **compactación** (defragmentación).



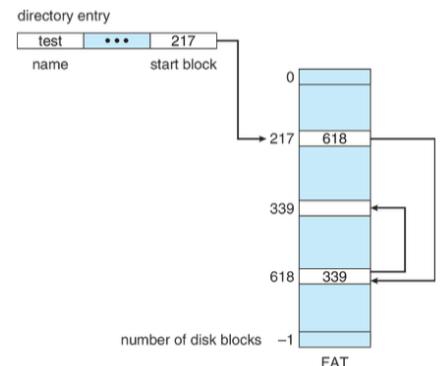
Asignación enlazada (“lista ligada”)

- Almacena punteros a bloque siguiente.
- Ejemplo: bloque de 512 Bytes, solo puede guardar 508 Bytes. Se pierde espacio.
- No requiere tamaño inicial de archivo. Puede crecer.
- Acceso solo secuencial (es más lento si quiero buscar algún elemento que este lejos del primero de la lista).
- Alternativa: almacenar *clusters* de bloques en lugar de bloques individuales (fragmentación interna).
- Vulnerable a falla de un bloque.



Asignación enlazada (“lista ligada”): FAT

- FAT: File Allocation Table.
- Tabla al inicio del disco con una entrada por archivo.
- Último cluster almacena EOF (*end of file*).
- Mejor acceso aleatorio.
- Limitado por tamaño de table (para FAT32, 4GB por archivo).



Asignación indexada

- Archivos contienen *index block*.
- *Index Block* contiene bloques del archivo.
- Último *cluster* almacena EOF (*end-of-file*).
- +Acceso aleatorio sin fragmentación Externa.
- Se pierde espacio por tamaño de *index block*.

Asignación indexada: Tamaño del *index block*

- **Esquema enlazado:** Última entrada de *index block* apunta a otro *index block*.
- **Índice multinivel:** Similar a tablas de página multinivel.
 - Ejemplo: blocks de 4KB, permiten 1024 punteros (de 32-bit).
 - Dos niveles permiten direccionar 1048576 bloques → 4 GB.
- **Esquema combinado:** Primeros p punteros son bloques directos, los siguientes apuntan a bloque de índice simple, los siguientes a bloque de índice doble, etc.

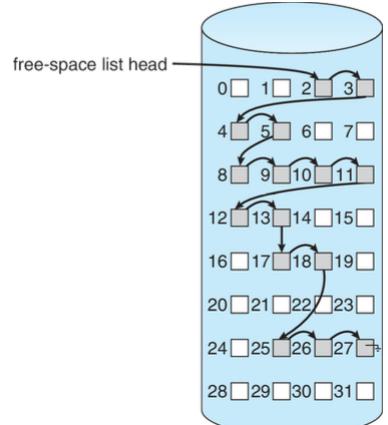
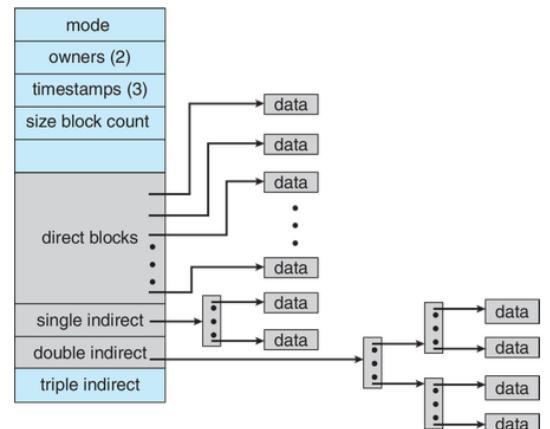
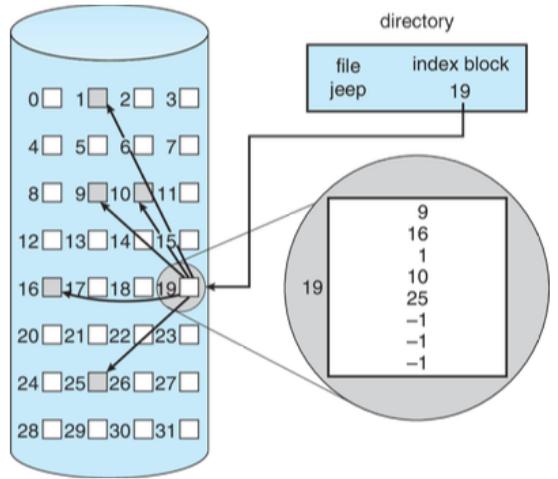
¿Cómo encontrar espacio libre?

Bit vector o **bitmap** de bloques libres.

- Sencillo, eficiente, e implementable en *hardware*.
- No escala tan bien.

Lista enlazada

- Los punteros apuntan a **bloques libres**.
- Bloques nuevos se agregan a la lista.
- Poco eficiente.
- Mejora: Agrupar espacios contiguos.
- Mantener punteros a espacios libres + número de espacios contiguos.



REDES

Infraestructura para permitir acceder a la red:

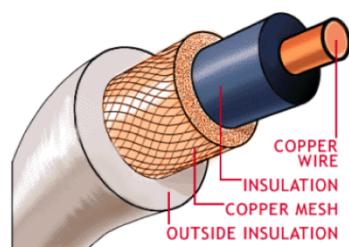
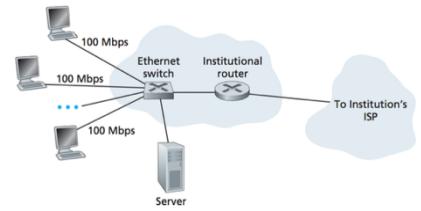
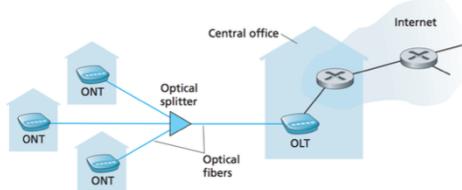
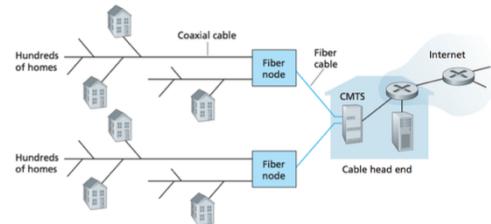
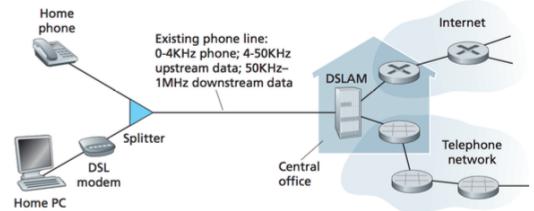
- **DIGITAL SUBSCRIBER LINE (DSL):**
 - DSL Módem utiliza línea telefónica tradicional.
 - DSL Módem convierte señal digital a análoga.
 - DSLAM: DSL Access Multiplexer.
 - DSLAM reconvierte señal análoga a digital.
- **CABLE**
 - Utiliza infraestructura de TV por cable.
 - Mezcla enlaces de fibra y coaxial
 - Utiliza **cable modem**.
- **FIBER: FTTH**
 - ONT: Optical Network Terminator.
 - OLT: Optical Line Terminator.
 - Conversión de señal óptica-eléctrica.
- **DIAL-UP, SATELLITE**
 - Dial-up: Enlace a través de línea telefónica
- **ETHERNET**
 - Estándar de conexión por pares trenzados de cobre.
 - Implementan **LAN (Local Area Network)**.
- **WI-FI**
 - Estándar de conexión inalámbrico de corto alcance.
 - Implementan **WLAN (Wireless LAN)**.

Par trenzado de cobre

- Cables trenzados se comportan como una antena.
- Uso para telefonía, ethernet, implementación de LANs.

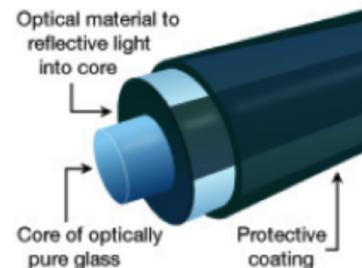
Cable coaxial de cobre

- Alta resistencia al ruido externo.
- Ancho de banda depende de la distancia.
- Fácil de modificar para insertar nodos nuevos.
- Medio compartido.



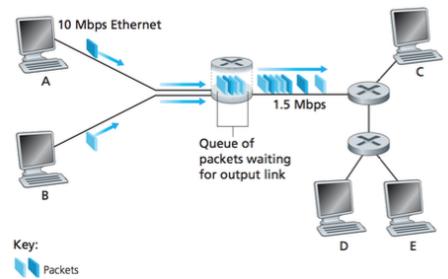
Fibra óptica

- Inmune a interferencia electromagnética.
- Cables hechos de fibra de vidrio.
- Alta velocidad de transferencia.
- Baja atenuación con la distancia.
- Monomodo: un haz de luz.
- Multimodo: múltiples haces a distintos ángulos.



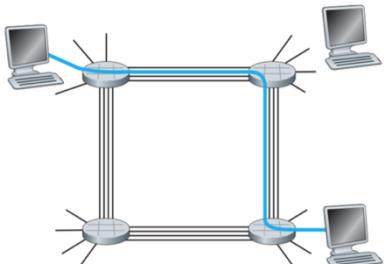
Packet Switching

- Dispositivos de red intercambian **paquetes**.
- Dispositivo (*router*) almacena paquetes (*store*).
- *Router* examina paquetes y determina punto de reenvío (*forward*).
- *Router* usa tablas de reenvío (*forwarding tables*)
- Tasa de transmisión determinada por el enlace más lento.



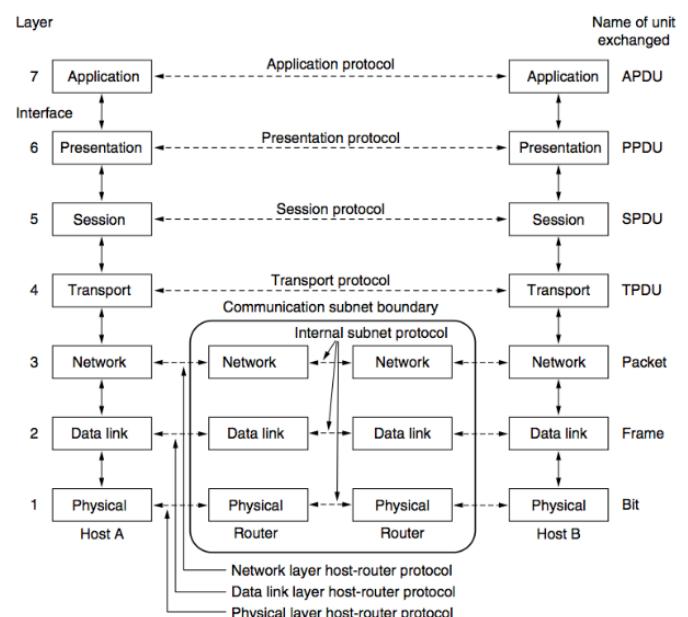
Circuit Switching

- Establecimiento previo a conexión entre origen-destino.
- Enfoque usado por comunicación **telefónica**.
- Sensible a saturación ante conexiones simultáneas.



Modelo de Capas

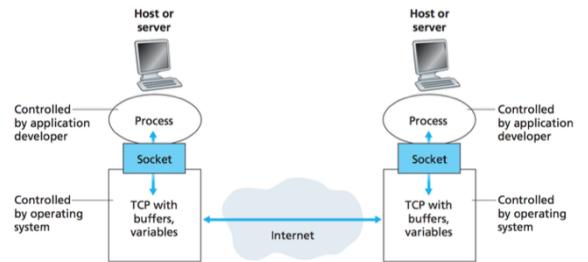
- **Aplicación:** Aplicaciones intercambian mensajes. HTTP.
- **Presentación:** Codificación, encriptación, serialización.
- **Sesión:** Establecimiento de sesión entre aplicaciones. Autorización y autenticación.
- **Transporte:** Mensaje de capa de aplicación en **segmentos**. TCP/UDP.
- **Red:** Servicio de entrega desde origen a destino.
- **Capa de enlace (link):** Paquetes se transmiten en cada par de **hosts** de acuerdo a las características del enlace.
- **Capa física:** Transmisión de **bits** a través de un medio. Protocolos para cada medio de transmisión.



Capa de aplicación

Socket:

- Elemento de software que interactúa con la red.
- Es una puerta de entrada/salida al sistema.
- Utiliza servicios de transporte (capa inferior).
- Solo existe en hosts de origen y destino (*end-nodes*).
- Es una interfaz entre el programador y la red.



DNS (DOMAIN NAME SYSTEM)

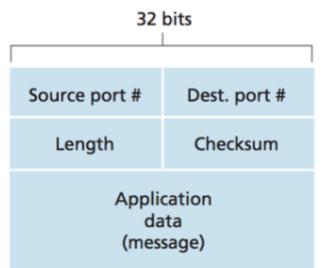
Se encarga de traducir los nombres de dominio legibles para los humanos en direcciones IP numéricas, que son utilizadas por las computadoras para localizar recursos en la red.

Cuando un usuario escribe una dirección URL o un nombre de dominio en el navegador web, el navegador envía una solicitud al servidor DNS local para obtener la dirección IP asociada con ese nombre de dominio.

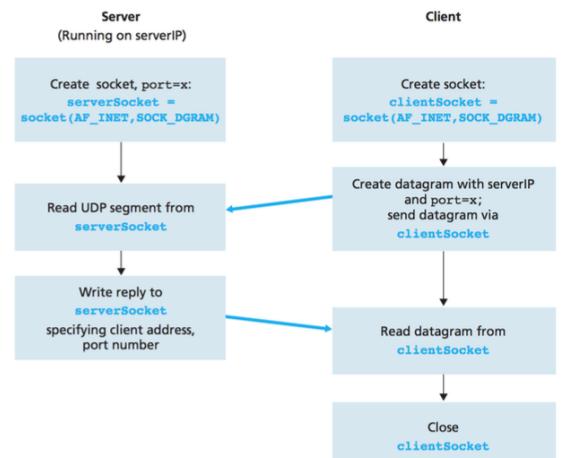
Ejemplo: www.google.com → 172.217.6.196

Conexión UDP (User Datagram Protocol)

1. Creación de *socket*.
2. Especificación de la dirección y el puerto de destino en la cabecera.
3. División en segmentos más pequeños llamados **datagramas**.
4. Los datagramas se envían a través de la red hacia la dirección IP y el número de puerto de destino especificados en la cabecera.
5. Recepción en el destino. Cuando llega un datagrama, el sistema operativo lo entrega al socket correspondiente.
6. Si los datos se dividen en varios datagramas, el sistema operativo del receptor ensambla los datagramas para reconstruir los datos originales en el orden correcto, utilizando la información de la cabecera para identificarlos.
7. Entrega a la aplicación receptora: Finalmente, los datos completos se entregan a la aplicación receptora, que puede procesarlos y responder si es necesario.

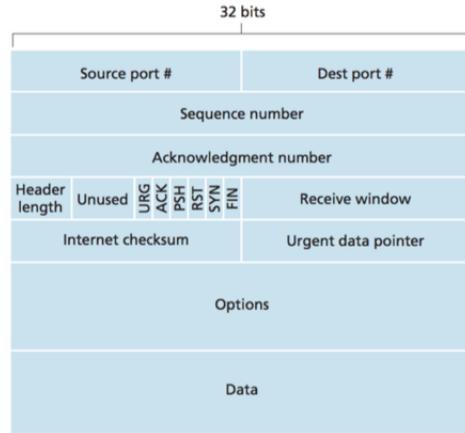


- Aplicación puede controlar directamente el envío.
- No hay demora en establecer conexión.
- No se reservan recursos.
- Menos *overhead* de espacio: 8 Bytes.
- Paquetes pueden no llegar.

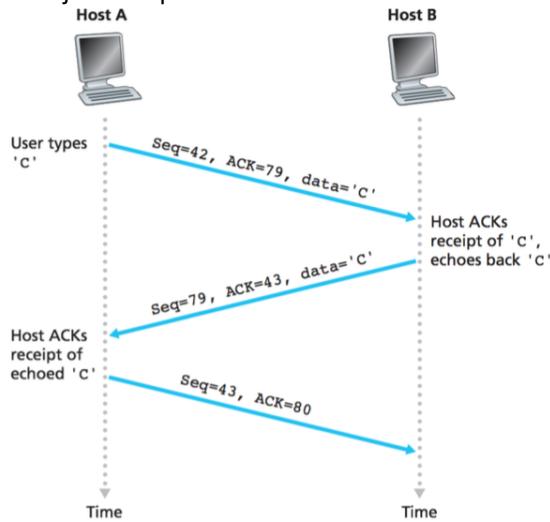


Conexión TCP (*Transmission Control Protocol*)

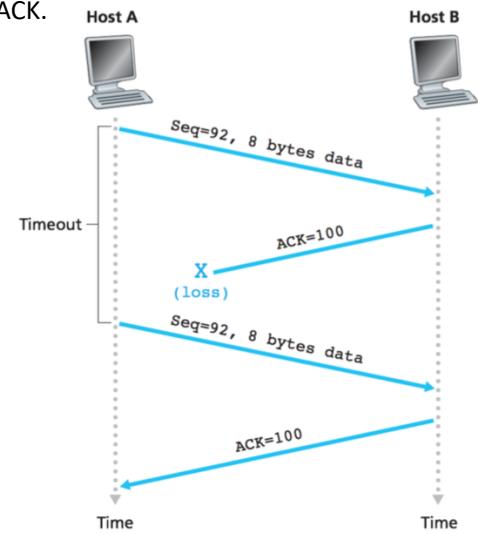
- Protocolo de transmisión fiable.
- Transmisión encadenada.
- Control de flujo a través del tamaño de la ventana.
- Buffer en emisor y en receptor.
- Orientado a conexión.



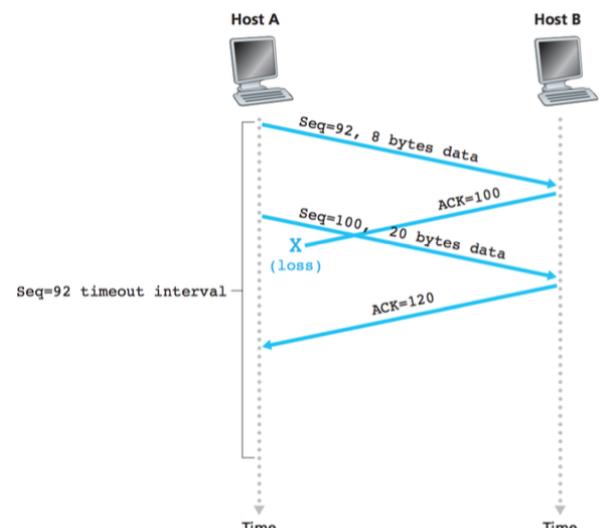
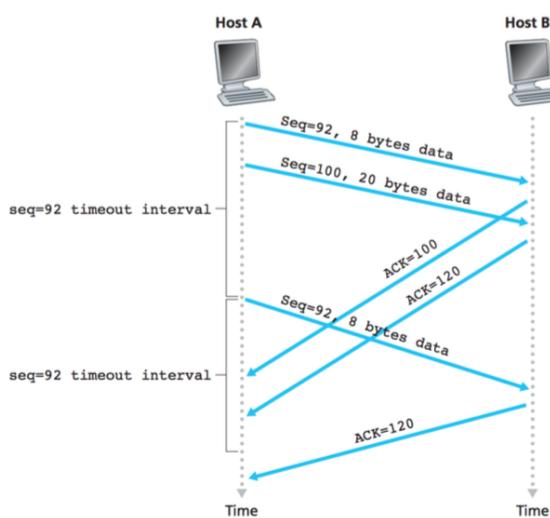
- Números de secuencia además cuentan bytes en la secuencia.
- ACK contiene el *sequence number* del siguiente byte que espera.
- Mensaje de respuesta funciona también como ACK.



- Si el ACK se pierde, se produce *timeout*.
 - A reenvía el paquete más antiguo sin ACK ($Seq=92$), y reinicia el *timer*.
 - A recibe ACK y reinicia timer, por $Seq=100$
 - B recibe $Seq=92$ duplicado. Lo descarta, pero reenvía ACK.

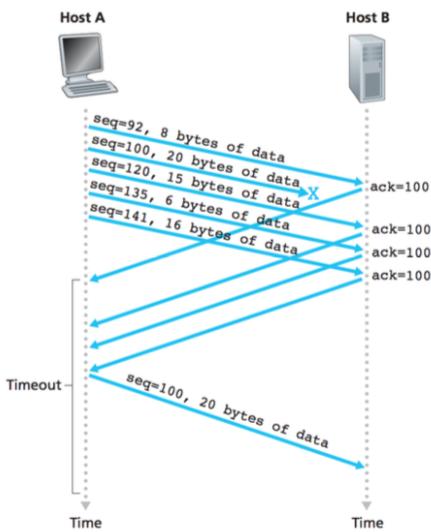


- Timeout ocurre antes que llegue el ACK.
 - A reenvía el paquete más antiguo sin ACK ($Seq=92$), y reinicia timer ($Seq=92$).
 - A recibe ACK y reinicia timer, por $Seq=100$
 - A NO reenvía $Seq=100$, ya que su ACK llega antes del timeout.
 - B recibe $Seq=92$ duplicado. Lo descarta, pero envía (cumulative) ACK.
- Cumulative ACK evita retransmisión.
 - El ACK=120 (ACK de $Seq=100$) llega antes del timeout de $Seq=92$.
 - Cumulative ACK (ACK=120) indica que se ha recibido el $Seq=92$ y el $Seq=100$.
 - A NO reenvía $Seq=92$.
 - A incrementa SendBase.



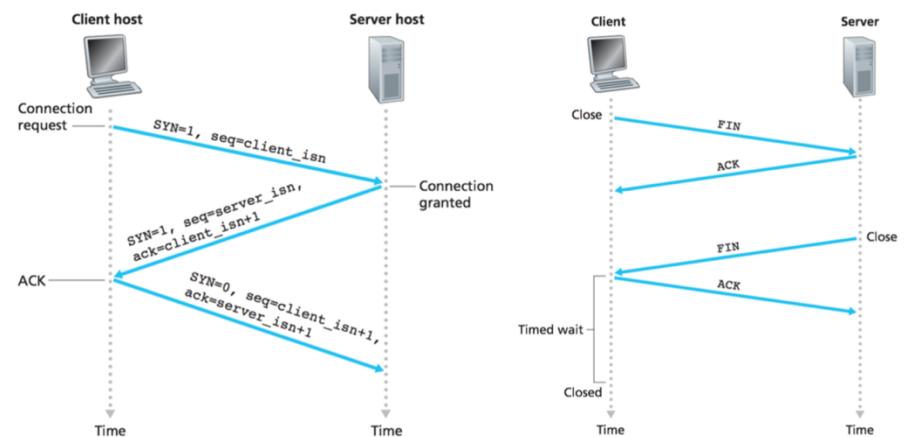
TCP: Retransmisión rápida

- *Timeouts* pueden tomar mucho tiempo.
- Retransmisión rápida envía proactivamente segmentos probablemente perdidos, antes que ocurra un *timeout*.



TCP: Handshake Protocol

- **Paso 1:** Cliente envía segmento SYN al servidor.
 - Incluye número de secuencia inicial (de cliente).
- **Paso 2:** Servidor recibe SYN, y responde con SYN ACK.
 - Servidor asigna buffer.
 - Servidor establece número de secuencia inicial (del servidor).
- **Paso 3:** Cliente recibe SYN ACK, y responde con ACK.
 - Este paquete ya puede contener datos.



DIRECCIONAMIENTO DE RED

¿Cómo un mensaje encuentra su destinatario?

- Gracias al **datagrama (paquete) IP**

Cada enlace puede tener distintas MTU (*Maximum Transfer Unit*)

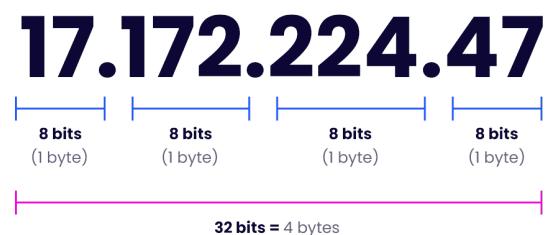
Al encontrar enlace con menor MTU.

- Router fragmenta el datagrama.
- Router envía fragmentos con mismo identificador.
- Router en el otro extremo del enlace lo reensambla.

Direcciones de 32 Bits = 4 Bytes (IPv4)

- $2^{32} = 4.294.967.296$ direcciones.
- **Dotted-decimal notation:** Grupo de 8 Bits (0 a 255).
- Dirección **única globalmente** (casi).

32 bits					
Version	Header length	Type of service	Datagram length (bytes)		
16-bit Identifier	Flags	13-bit Fragmentation offset			
Time-to-live	Upper-layer protocol	Header checksum			
32-bit Source IP address					
32-bit Destination IP address					
Options (if any)					
Data					



Ejemplo:



- IP: **10010010.10011011.00001101.01001110**
- Máscara: **1111111.11111111.11111111.00000000**
- Subred: **10010010.10011011.00001101.00000000**

DHCP: *Dynamic Host Configuration Protocol*

Protocolo de red utilizado para asignar automáticamente direcciones IP y otros parámetros de configuración de red a dispositivos en una red. El DHCP facilita la gestión eficiente de direcciones IP y simplifica la configuración de dispositivos en una red, especialmente en redes de gran tamaño.

NAT: *Network Address Translation*

Técnica utilizada en redes de computadoras para permitir que varios dispositivos en una red privada compartan una única dirección IP pública para acceder a Internet. NAT se utiliza comúnmente en enrutadores domésticos y empresariales para permitir que múltiples dispositivos se conecten a Internet a través de una sola dirección IP pública.

La principal diferencia que tiene **IPv6** y no **IPv4** es el tamaño de la dirección IP. Mientras que IPv4 utiliza direcciones IP de 32 bits, IPv6 utiliza direcciones IP de 128 bits. Esta diferencia en el tamaño de la dirección IP tiene implicaciones significativas en la cantidad de direcciones únicas disponibles en cada versión del protocolo de Internet.

Algoritmos de routing

• **Link-State (LS)**

- Cada nodo envía su información de conectividad (vecinos y costos) a sus vecinos.
- Información se comunica a los demás vía *flooding*.
- Cuando todos tienen información de la topología, cada uno calcula las rutas más cortas.
- Algoritmo de Dijkstra (1957).

- **Distance Vector (DV):**

- Cada nodo mantiene una tabla de la mejor ruta a sus vecinos.
- Cada nodo comparte su tabla con sus vecinos.
- Cuando la información se ha propagado a todos los nodos, cada uno conoce las mejores rutas.
- Diferencia con LS: Se propaga información de rutas en lugar de enlaces.

Link State

- N nodos, E enlaces: $O(NE)$ mensajes
- $O(N^2)$ complejidad en cada nodo
- Ante errores, se propaga ausencia de enlace local.

Distance Vector

- Se ejecuta en paralelo y solo se propaga la ruta a los vecinos.
- Ante errores, se propaga el error.
- Puede converger muy lentamente. Tiempo de convergencia indeterminado.

¿Cómo saber si un *frame* llegó con errores?

- Se agregan bits de redundancia que ayuden a detectar algunos errores.
- **Checksum:** Este método implica que el dispositivo emisor calcule una suma de verificación (*checksum*) basada en los datos del *frame* y la incluye en el encabezado del *frame*. Al llegar al dispositivo receptor, se realiza nuevamente el cálculo de la suma de verificación. Si la suma calculada en el receptor coincide con la suma enviada en el *frame*, se asume que el *frame* llegó sin errores.
- **CRC (Cyclic Redundancy Check):** En este método, el dispositivo emisor y el receptor utilizan un polinomio generador (*checksum*) predefinido para calcular un valor CRC a partir de los datos del *frame*. El valor CRC se coloca en el *frame* y se envía junto con los datos. Al llegar al receptor, se recalcula el valor CRC utilizando el mismo polinomio generador. Si el valor CRC calculado coincide con el valor CRC recibido, se asume que el *frame* llegó sin errores. De lo contrario, se considera que el *frame* tiene errores.

SSL (Secure Sockets Layer): protocolo de seguridad que proporciona una capa adicional de encriptación y autenticación para proteger la comunicación entre un cliente y un servidor en Internet.

TLS (Transport Layer Security): Se implementa en la capa de transporte del modelo OSI y se utiliza para proteger una amplia variedad de servicios en línea, incluyendo el acceso a sitios web (HTTPS), correos electrónicos seguros, VPNs y más.

Firewall: Barrera de seguridad que se utiliza para proteger una red de computadoras o un dispositivo de posibles amenazas y accesos no autorizados desde Internet u otras redes. Su función principal es controlar el tráfico de red entrante y saliente, permitiendo o bloqueando ciertos tipos de datos y conexiones, con el objetivo de proteger los recursos y datos de la red y garantizar la privacidad y seguridad de los usuarios.