

## 1.1 . ¿ Qué es un sistema operativo?

El sistema operativo es el principal programa que se ejecuta en toda computadora de propósito general.

Los hay de todo tipo, desde muy simples hasta terriblemente complejos , y entre más casos de uso hay para el cómputo en la vida diaria , más variedad habrá en ellos.

El ambiente gráfico , los programas que se ejecutan en éste , los lenguajes de programación en los cuales están desarrollados y en que más fácilmente se puede desarrollar para ellos , e incluso el conjunto básico de funciones que las bibliotecas base ofrecen son principalmente clientes del sistema operativo - se ejecutan sobre él , y ofrecen sus interfaces a los usuarios (incluidos, claro, los desarrolladores).

### 1.1.1. ¿Por qué estudiar los sistemas operativos?

Como desarrolladores , comprender el funcionamiento básico de los sistemas operativos y las principales alternativas que ofrecen en muchos de sus puntos , o saber diseñar algoritmos y procesos que se ajusten mejor al sistema operativo en que vayan a ejecutarse , puede resultar en una diferencia cualitativa decisiva en el producto final.

## 1.2. Funciones y objetivos del sistema operativo

El sistema operativo es el único programa que interactúa directamente con el hardware de la computadora. Sus funciones primarias son:

El sistema operativo se encarga de proporcionar una serie de abstracciones para que los programadores puedan enfocarse en resolver las necesidades particulares de sus usuarios . Un ejemplo de tales abstracciones es que la información está organizada en archivos y directorios (en uno o muchos dispositivos de almacenamiento).

Un sistema de cómputo puede tener a su disposición una gran cantidad de recursos (memoria, espacio de almacenamiento, tiempo de procesamiento, etc.) , y los diferentes procesos que se ejecuten en él compiten por ellos.

En un sistema multiusuario y multitarea cada proceso y cada usuario no tendrá que preocuparse por otros que estén usando el mismo sistema -Idealmente , su experiencia será la misma que si el sistema estuviera exclusivamente dedicado a su atención (aunque fuera un sistema menos poderoso) .

Para implementar correctamente las funciones de aislamiento hace falta que el sistema operativo utilice hardware específico para dicha protección.

### 1.3.3 Sistemas multi programados

La programación multitareas o los sistemas multiprogramados buscaban maximizar el tiempo de uso efectivo del procesador ejecutando varios procesos al mismo tiempo.

Un proceso no debe sobreescribir el espacio de memoria de otro (ni el código , ni los datos), mucho menos el espacio del monitor . Esta protección se encuentra en la Unidad de Manejo de Memoria (MMU) , presente en todas las computadoras de uso genérico desde los años noventa.

Ciertos dispositivos requieren bloqueo para ofrecer acceso exclusivo / único: cintas e impresoras, por ejemplo, son de acceso estrictamente secuencial , y si dos usuarios intentan usarlas al mismo tiempo, el resultado para ambos se corromperá. Para estos dispositivos, el sistema debe implementar otros spools y mecanismos de bloqueo.

## SO Procesos 01 (Video)

### Sistema de computación monoprocesador

Un monoprocesador, como su nombre lo indica, es un procesador que solo puede ejecutar un proceso a la vez. Esto quiere decir que si se requiere que se ejecuten varias tareas al mismo tiempo, no va a ser posible que se realicen con satisfacción. Lo que pueden hacer los monoprocesadores es alternar las tareas. Es capaz de manejar solamente un procesador de la computadora , de manera que si la computadora tuviese más de uno le sería inútil. El ejemplo más típico de este tipo de sistemas es el DOS y MacOS.

### Conexion Multiusuario

Se considera entonces como Multiusuario a todo sistema operativo que permite a un mínimo de dos usuarios a poder acceder a distintas categorías dentro del equipo, compartiendo el mismo entorno físico pero con distintos accesos que son gestionados por aquel que tiene la jerarquía de administrador.

Lo más conocido sin lugar a dudas es la tecnología difundida por el sistema de Microsoft Windows , que desde su versión denominada como Windows 95 permite personalizar cada uno de los Usuarios, administrar las distintas Sesiones de Usuario y asignar distintos accesos y hasta la posibilidad de personalizar cada Interfaz Gráfica para cada una de ellas, lo que evolucionó posteriormente con las últimas versiones de Windows Server con la ejecución de sesiones de usuario mediante un Acceso Remoto.

### Multiprogramación concurrente de procesos

Se denomina multiprogramación a una técnica por la que dos o más procesos pueden alojarse en la memoria principal y ser ejecutados concurrentemente por el procesador o CPU.

Con la multiprogramación, la ejecución de los procesos (o hilos) se va solapando en el tiempo a tal velocidad, que causa la impresión de realizarse en paralelo (simultáneamente). Se trata de un paralelismo simulado, dado que la CPU sólo puede trabajar con un proceso cada vez (el proceso activo). De ahí que, en rigor, se diga que la CPU ejecuta «concurrentemente» (no simultáneamente) varios procesos; en un lapso de tiempo determinado, se ejecutarán alternativamente partes de múltiples procesos cargados en la memoria principal.

en un sistema multiprogramado, cuando un proceso  $P_x$  concluye o se bloquea (en espera de una operación de E/S), el núcleo del sistema operativo toma el control de la CPU para efectuar lo que se denomina un «cambio de contexto», a fin de dar turno a otro proceso  $P_y$  para que se ejecute.

## Relación entre proceso y recurso es Fuertemente Acoplado

A los sistemas fuertemente acoplados se les conoce como sistemas multiprocesadores, y todos los procesadores que lo forman pueden utilizar todos los recursos del sistema.

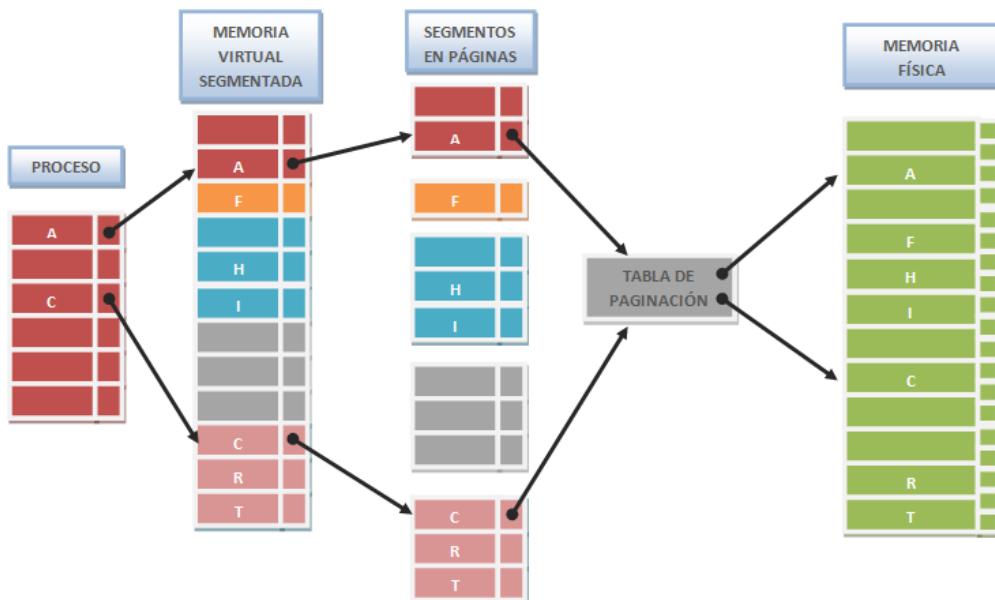
Sus características más importantes son:

- 1) Memoria común.** Todos los procesadores del sistema pueden acceder a una memoria principal común, aunque cada uno de ellos pueda también tener una memoria de datos propia.
- 2) Entrada/Salida.** Todos los procesadores del sistema comparten el acceso a los dispositivos de entrada/salida.
- 3) Sistema operativo común.** El sistema se controla mediante un sistema operativo, que regula las interacciones entre procesadores y programas. Los sistemas fuertemente acoplados deben disponer de un mecanismo de sincronización entre procesadores. En general, todos los procesadores deben ser iguales, formando así una configuración simétrica.

En este tipo de sistemas es posible que los procesadores originen conflictos en el acceso a memoria principal o a los dispositivos I/O. Estos conflictos deben ser minimizados por el sistema operativo y por la estructura de interconexión.

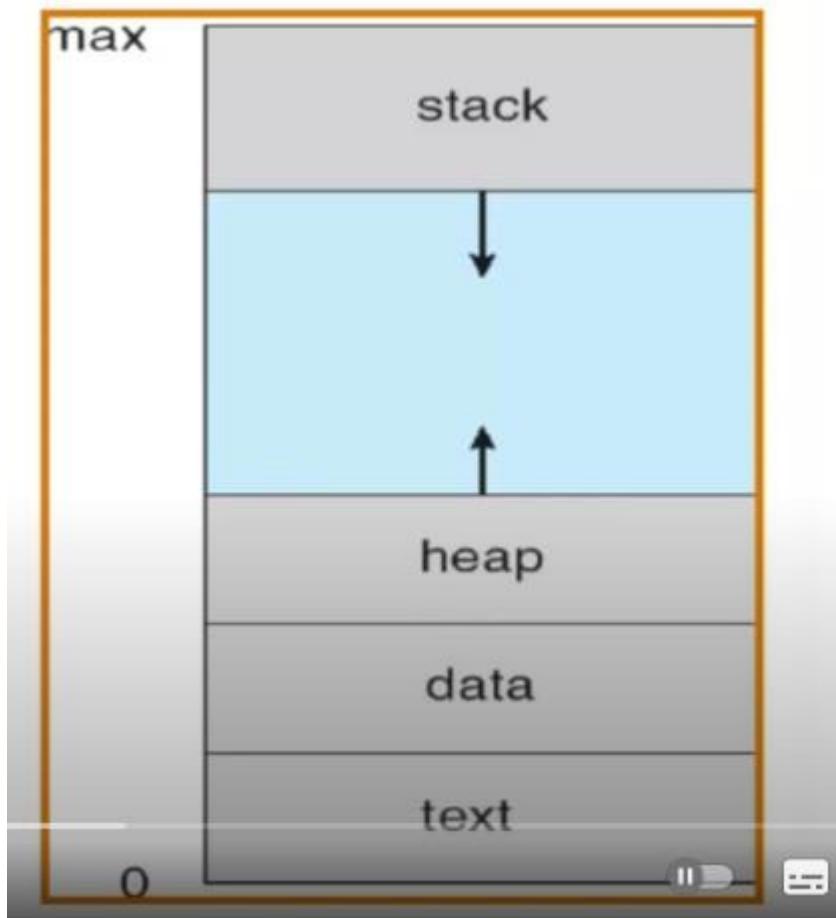
## Memoria lógica / Memoria física

	DIRECCIÓN LÓGICA	DIRECCIÓN FÍSICA
Qué es?	Es la dirección virtual generada por la CPU	La dirección física es una ubicación en una unidad de memoria.
Espacio de dirección	El conjunto de todas las direcciones lógicas generadas por la CPU en referencia a un programa se denomina Espacio de direcciones lógicas.	El conjunto de todas las direcciones físicas asignadas a las direcciones lógicas correspondientes se denomina Dirección física.
Visibilidad	El usuario puede ver la dirección lógica de un programa.	El usuario nunca puede ver la dirección física del programa
Acceso	El usuario usa la dirección lógica para acceder a la dirección física.	El usuario no puede acceder directamente a la dirección física.
Generación	La dirección lógica es generada por la CPU	La dirección física es calculada por MMU



## Proceso en memoria lógica

# Proceso en memoria lógica



**0** - Dirección de memoria más baja (A partir de donde está almacenado el proceso).

**Max** - Dirección de memoria más alta (A partir de donde está almacenado el proceso).

**Text** - Código del programa (instrucciones en lenguaje de máquina).

**Data** - Se almacenan los datos estáticos (vectores, matrices, variables simples del programa).

**Heap** - Se almacenan datos dinámicos (Listas enlazadas, Árboles, pilas, colas, etc.).

**Stack** - Se almacena las direcciones de retorno cuando se llaman a ejecutar funciones del usuario desde el programa principal.

El sistema operativo es un **sistema de gestión de recursos de un sistema de computación**. Administra los recursos de un sistema de computación para permitir que los procesos los **usen ordenadamente sin que el sistema entre en ningún fallo** y además que de una respuesta óptima del sistema. Tal cual lo hace que cualquier sistema de gestión administrativo, todos los sistemas necesitan para administrar datos y procesos.

- Espacio de direcciones
- Hijo duplicado del padre
- Se carga un programa en el hijo
  - Ejemplos de llamadas al sistema en UNIX
- fork** para crear nuevos procesos
- exec**, después de **fork** reemplaza el espacio del proceso con un programa nuevo

## Creación de proceso

Los SO proveen mecanismos para que los **procesos puedan crear otros procesos**

→ Llamada al sistema; El proceso de creación se puede repetir recursivamente creándose una “**estructura familiar**” → Árbol de procesos, asignando los recursos al nuevo proceso de la siguiente forma:

- Los obtiene directamente del SO
- El **padre debe repartir sus recursos con el proceso hijo** o compartir todos o parte de ellos con él y así se evita que un proceso bloquee el sistema multiplicándose indefinidamente.
- En términos de ejecución el **padre continúa ejecutándose en paralelo con su/s hijo/s** y el padre **espera a que alguno o todos sus hijos hayan terminado**
- En términos del espacio en memoria, el proceso hijo es un clon del proceso padre y tiene ya un programa cargado en memoria

## Creación de proceso

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    /* fork: un proceso hijo */
    pid = fork();

    if (pid < 0) { /* Error */
        fprintf(stderr, "Falló el fork");
        exit(-1);
    } else if (pid == 0) { /* Proceso hijo */
        execvp("/bin/ls", "ls", NULL);
    } else { /* proceso padre */
        wait(NULL);
        printf("Concluyó hijo\n");
        exit(0);
    }
}
```

## Estado de un proceso

**nuevo:** El proceso está siendo creado.

**corriendo:** Se están ejecutando instrucciones.

**esperando:** El proceso está esperando que ocurra un evento.

**listo:** El proceso está esperando ser asignado a un procesador.

**terminado:** El proceso ha terminado su ejecución.

## Terminación de proceso

El proceso ejecuta su último enunciado y le pide al sistema operativo que lo borre (**exit**):

- Datos de salida de hijo a padre (**via wait**)
- Recursos del proceso son liberados por el SO

Padre puede terminar la ejecución de hijo (**abort**) si :

- El hijo excedió los recursos asignados
- La tarea asignada al hijo ya no es necesaria
- Si el padre está terminando

Algunos sistemas operativos no permiten a los hijos continuar trabajando

Todos los hijos son terminados - **terminación en cascada**

## SO Procesos 02

```
salo@DESKTOP-0M99BC2:~$ ps
 PID TTY          TIME CMD
 10 tty1        00:00:00 bash
 16 tty1        00:00:00 ps
salo@DESKTOP-0M99BC2:~$
```

**PID** - identificador único del proceso

**TTY** - terminal virtual del proceso

**TIME** - tiempo acumulado de ejecución del proceso

**CMD** - nombre del comando

```
salo@DESKTOP-0M99BC2:~$ ps
 PID TTY          TIME CMD
 10 tty1        00:00:00 bash
 16 tty1        00:00:00 ps
salo@DESKTOP-0M99BC2:~$ ps -l
F S  UID    PID  PPID   C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 1000     10      9  0  80    0 -  3785 -          tty1        00:00:00 bash
0 R 1000     17     10  0  80    0 -  4415 -          tty1        00:00:00 ps
salo@DESKTOP-0M99BC2:~$
```

**PID** - Process ID

### PPID - Parent Process ID

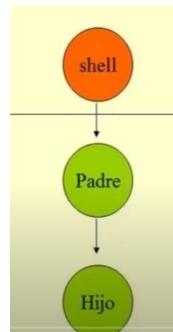
Quiere decir que el proceso ‘ps’ (17) fue **creado** por el proceso ‘bash’ (10).  
El proceso ‘bash’ está **sleeping (S)** mientras el proceso ‘ps’ está **running (R)**.

Si el comando ‘gedit’ no trabaja, usar comando ‘**export DISPLAY=0:0**’ y luego intentar ‘**gedit**’ de nuevo.

## Creación de procesos en linux (fork)

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

main()
{
    fork();
    exit(0);
}
```



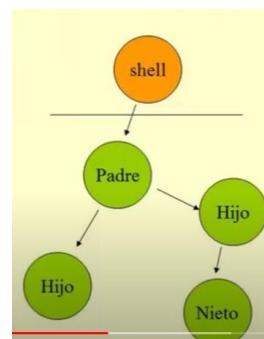
Los círculos representan los procesos que se estaría ejecutando con ese código

**Shell** - Permite ejecutar al usuario sus aplicaciones

**Círculos verde** - procesos que se crean debajo del **Shell**, **Padre** fue creado por **Shell** y **Hijo** fue creado por **Padre**

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

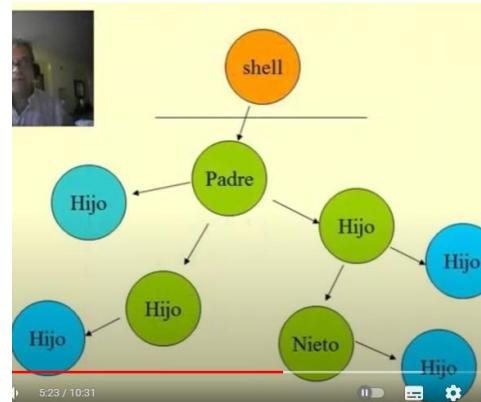
main()
{
    fork();
    fork();
    exit(0);
}
```



Aca el **Shell** crea un fork, y luego el **Padre** crea un **Hijo**, y luego los dos crean un fork cada uno

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

main()
{
    fork();
    fork();
    fork();
    exit(0);
}
```



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

main(int argc, char * argv[])
{
    pid_t pid;
    int n = atoi(argv[1]);
    int i ;
    for(i = 0; i < n ; i++)
    {
        pid = fork(); // Red arrow points here
        if (pid > 0 ) break;
    }
    printf("Padre %d saliendo ---- creo hijo %d\n",getpid(),pid);
    exit(0);
}
```



Este es el código para la tabla de abajo



## C Linux Procesos 01

### Creación de procesos con fork

fork es una función que embebe una llamada al sistema operativo que permite la creación de un proceso.

**Manual del Programador de Linux**

#### NOMBRE

fork - crean un proceso hijo

#### SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork ( void );
```

#### DESCRIPCIÓN

fork crea un proceso hijo que difiere de su proceso padre sólo en su PID y PPID , y en el

hecho de que el uso de recursos esté asignado a 0. Los candados de fichero ( file locks ) y las señales pendientes no se heredan .

En linux , fork está implementado usando páginas de copia-en-escritura (copy-on-write), así que la única penalización en que incurre fork es en el tiempo y memoria requeridos para duplicar las tablas de páginas del padre , y para crear una única estructura de tarea ( task structure ) para el hijo .

### VALOR DEVUELTO

En caso de éxito, se devuelve el PID del proceso hijo en el hilo de ejecución de su padre , y se devuelve un 0 en el hilo de ejecución del hijo. En caso de fallo, se devolverá un -1 en el contexto del padre, no se creará ningún proceso hijo , y se pondrá en errno un valor apropiado.

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo-1.c -o ejemplo-1
```

**cc** - compilador de c#

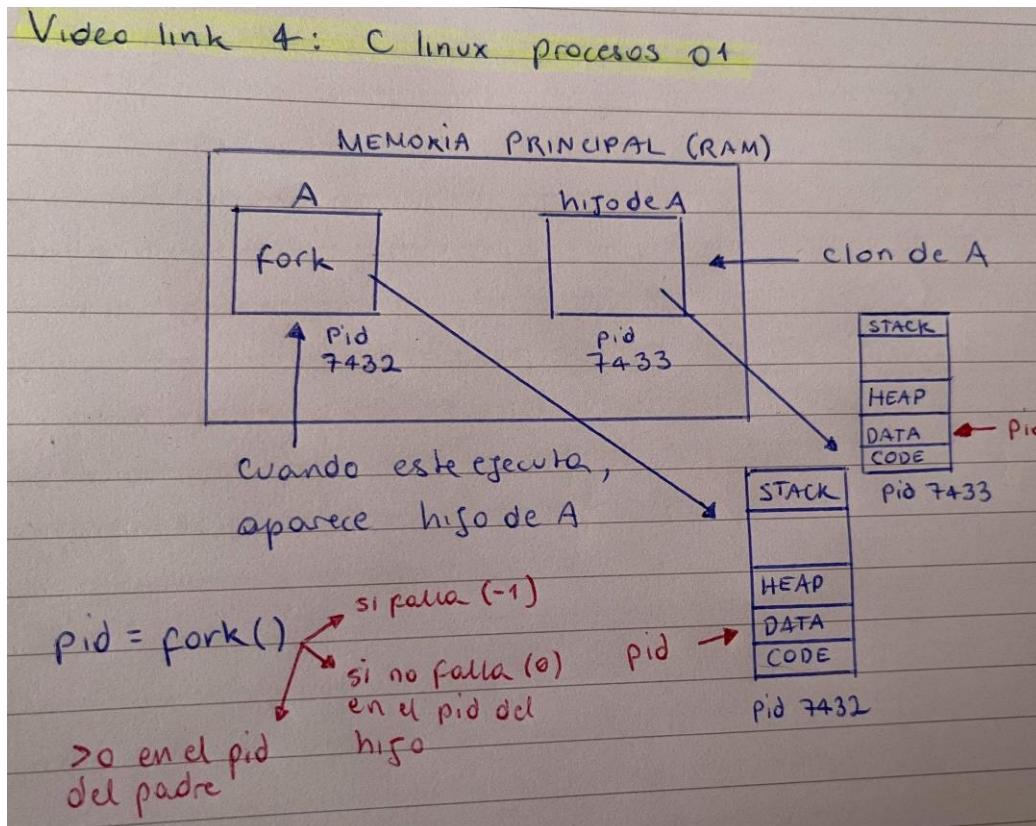
**ejemplo-1.c** - programa fuente a compilar

**-o** - salida del proceso de compilación nos devuelva...

**ejemplo-1** - el programa ejecutable que se va a llamar '**ejemplo-1**'

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo-1.c -o ejemplo-1
salo@DESKTOP-0M99BC2:~$ ./ejemplo-1
Quien escribio este texto en pantalla?
Quien escribio este texto en pantalla?
```

**./ejemplo-1** - ejecutar ejemplo-1



**pid de A (padre)** - la variable en el área del DATA del **Padre** tiene 7433 (hijo)

**pid del hijo de A (hijo)** - la variable en el área del DATA del **Hijo** tiene 0

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 int main(void)
7 {
8     pid_t pid; // declarar una variable de tipo 'pid_t'
9     pid = fork(); // llamar a la funcion para que ejecute la llamada al sistema y retorna un dato
    tipo pid_t
10    if (pid == 0)
11        printf("Yo soy el hijo\n"); //hijo
12    else
13        if (pid > 0)
14            printf("Yo soy el padre\n"); // padre
15        else // osea numero negativo
16            printf("fork fallo\n");
17    return 0; // retorne la funcion principal (main)
18 }
```

```

salo@DESKTOP-0M99BC2:~$ cc ejemplo-2.c -o ejemplo-2
salo@DESKTOP-0M99BC2:~$ ./ejemplo-2
Yo soy el padre
Yo soy el hijo
salo@DESKTOP-0M99BC2:~$
```

Para **copiar** un texto de gedit a uno nuevo:

```
salo@DESKTOP-0M99BC2:~$ cp ejemplo-2.c ejemplo-3.c  
salo@DESKTOP-0M99BC2:~$
```

**cp ejemplo-2.c** - copiar lo que está en ejemplo-2.c

**ejemplo-3.c** - nuevo texto en gedit con lo copiado del ejemplo 2

```
1 #include <sys/types.h>  
2 #include <unistd.h>  
3 #include <stdio.h>  
4 #include <stdlib.h>  
5  
6 int main(void)  
7 {  
8     pid_t pid; // declarar una variable de tipo 'pid_t'  
9     pid = fork(); // llamar a la función para que ejecute la llamada al sistema y retorna un dato tipo pid_t  
10    if (pid == 0)  
11        printf("Soy el hijo %d mi padre es %d\n",getpid(),getppid()); // hijo  
12    else  
13        if (pid > 0)  
14            printf("Soy el padre %d y mi hijo es %d y mi padre es %d\n",getpid(),pid,getppid()); // padre  
15        else // osea numero negativo  
16            printf("fork fallo\n");  
17    return 0; // retorne la función principal (main)  
18 }
```

**getpid()** - te da el número de Process ID

**getppid()** - te da el número de Process ID del Padre

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo-3.c -o ejemplo-3  
salo@DESKTOP-0M99BC2:~$ ./ejemplo-3  
Soy el padre 1076 y mi hijo es 1077 y mi padre es 811  
Soy el hijo 1077 mi padre es 1076  
salo@DESKTOP-0M99BC2:~$
```

**Padre del Padre (bash) pid** - 881 (Shell)

**Padre pid** - 1076

**Hijo pid** - 1077

Si aparece el número “1” en el **getppid()** quiere decir que el proceso **padre terminó primero** y el hijo quedó huérfano sin proceso padre, el número **1 es el PID del padrino** y se convierte en PPID de todos los procesos hijos huérfanos.

```
jromer@debianPGhost1:~/tallerC/procesos$ soy el hijo 2790 mi padre es 1
```

Cómo **evitar tener procesos huérfanos** en el siguiente video

## C Linux Procesos 02

### Sincronización y comunicación entre proceso Padre e Hijo con wait

wait es una función que embebe una llamada al sistema operativo que permite sincronizar la terminación de un proceso hijo, haciendo que el proceso padre pueda esperar la terminación de un proceso hijo.

#### NOMBRE

wait , waitpid - espera por el final de un proceso

#### SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait ( int * status ) ;
pid_t waitpid ( pid_t pid , int * status , int options ) ;
```

#### DESCRIPCIÓN

La función wait suspende la ejecución del proceso actual hasta que un proceso hijo ha terminado, o hasta que se produce una señal cuya acción es terminar el proceso actual o llamar a la función manejadora de la señal . Si un hijo ha salido cuando se produce la llamada ( lo que se entiende por proceso " zombie " ) , la función vuelve inmediatamente . Todos los recursos del sistema reservados por el hijo son liberados .

La función **waitpid** suspende la ejecución del proceso en curso hasta que un hijo especificado por el argumento **pid** ha terminado , o hasta que se produce una señal cuya acción es finalizar el proceso actual o llamar a la función manejadora de la señal .

Si el hijo especificado por **pid** ha terminado cuando se produce la llamada ( un proceso " zombie " ) , la función vuelve inmediatamente . Todos los recursos del sistema reservados por el hijo son liberados .

```
1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6
7 int main(void)
8 {
9     pid_t pid , hijoterm; // declarar una variable de tipo 'pid_t'
10    int status;
11    pid = fork(); // llamar a la función para que ejecute la llamada al sistema y retorna un dato tipo pid_t
12    if (pid == 0)
13    {
14        printf("Soy el hijo %d mi padre es %d\n",getpid(),getppid());
15        sleep(5); // para poder ver el efecto de espera por 5 segundos
16        printf("Termino ya\n");
17        exit(0);
18    }
19    else
20    {
21        if (pid > 0)
22        {
23            printf("Soy el padre %d y mi hijo es %d y mi padre es %d\n",getpid(),pid,getppid());
24            printf("Padre esperando la terminación del hijo\n");
25            hijoterm = wait(&status);
26        }
27        else // osea numero negativo
28            printf("fork fallo\n");
29    }
30 }
```

La comunicación entre el padre y el hijo es el 'wait(&status)' con el 'exit(0)'.

El parámetro del exit se para en el status del wait (espacio de direcciones en el padre).

Cuando el hijo termina y hace un exit, le lleva una señal al proceso padre y lo desbloquea, y además captura en el área de memoria asociado a la variable status el parámetro que se le asocia al exit.

```

7 int main(void)
8 {
9     pid_t pid_hijoterm; // declarar una variable de tipo 'pid_t'
10    int status;
11    pid = fork(); // llamar a la funcion para que ejecute la llamada al sistema y retorna un dato tipo
12    pid_t
13    if (pid == 0)
14    {
15        printf("Soy el hijo %d mi padre es %d\n",getpid(),getppid());
16        sleep(5); // para poder ver el efecto de espera por 5 segundos
17        printf("Termino ya\n");
18        exit(0);
19    }
20    else
21    {
22        printf("Soy el padre %d y mi hijo es %d y mi padre es %d\n",getpid(),pid,getppid());
23        printf("Padre esperando la terminacion del hijo\n");
24        hijoterm = wait(&status);
25        printf("Termino mi hijo %d\n",hijoterm);
26        printf("Mi hijo termino y me envio un %d\n",WEXITSTATUS(status)); //para ver el valor de
27        status
28    }
29    else // osea numero negativo
30        printf("fork fallo\n");
31    return 0; // retorne la funcion principal (main)
32 }
```

```

salo@DESKTOP-0M99BC2:~$ cc ejemplo-5.c -o ejemplo-5
salo@DESKTOP-0M99BC2:~$ ./ejemplo-5
Soy el padre 1315 y mi hijo es 1316 y mi padre es 811
Soy el hijo 1316 mi padre es 1315
Padre esperando la terminacion del hijo
Termino ya
Termino mi hijo 1316
Mi hijo termino y me envio un 0 ]
salo@DESKTOP-0M99BC2:~$
```

El WEXITSTATUS(status) también se escribe como (status/256).

## C Linux Procesos 03

### Dar sentido de existencia al proceso Hijo con exec

exec es una función de la familia de funciones exec que permite copiar la imagen binaria de un archivo ejecutable en el espacio de direcciones del proceso , en este caso proceso hijo , para que se convierta en un proceso distinto al proceso padre (en este caso).

### NOMBRE

exec , execlp , execle , execv , execvp - ejecutan un fichero

### SINOPSIS

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

```
extern char ** environ ;
```

```
int execl ( const char * path , const char * arg , ... ) ;
int execlp ( const char * file , const char * arg , ... ) ;
int execle ( const char * path , const char * arg , ... , char * const envp [ ] ) ;
int execv ( const char * path , char * const argv [ ] ) ;
int execvp ( const char * file , char * const argv [ ] ) ;
```

### DESCRIPCIÓN

La familia de funciones **exec** reemplaza la imagen del proceso en curso con una nueva .

El **EXEC** puede cambiar el proceso hijo (clon del padre) a otro proceso diferente. Puede cambiar la imagen binaria ejecutable del hijo, a otra imagen binaria ejecutable.

### Como se ve esto en un programa:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     printf("Soy el hijo convertido en otra cosa\n");
7     printf("Hola mundo cruel!\n");
8     exit(0);
9 }
```

Aca creamos un programa insignificativo llamado “otracosa.c”. Y esto es lo que muestra al ejecutar:

```
salo@DESKTOP-0M99BC2:~$ cc otracosa.c -o otracosa
salo@DESKTOP-0M99BC2:~$ ./otracosa
Soy el hijo convertido en otra cosa
Hola mundo cruel!
```

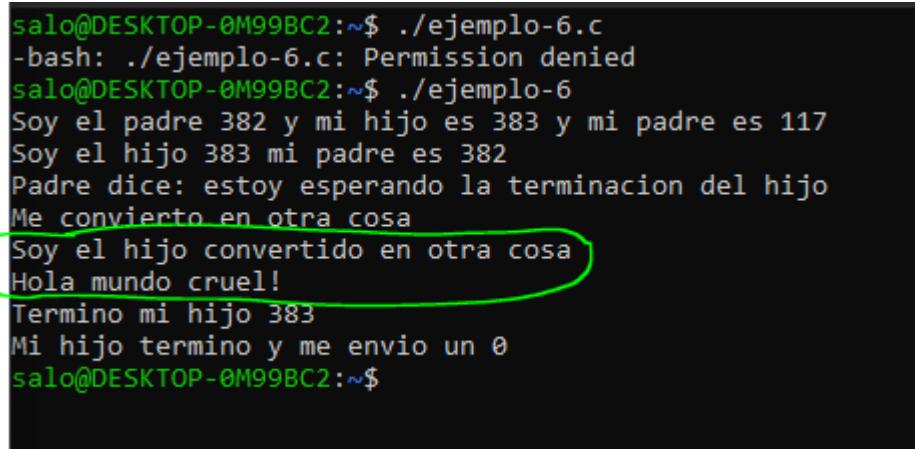
Ahora copio el ejemplo 5 (ejemplo-5.c) a un nuevo ejemplo (ejemplo-6.c) para modificar algunas cosas e implementar el **EXEC**.

```

1 #include <sys/types.h>
2 #include <unistd.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6
7 int main(void)
8 {
9     pid_t pid , hijoterm; // declarar una variable de tipo 'pid_t'
10    int status;
11    pid = fork(); // llamar a la funcion para que ejecute la llamada al sistema y retorna un dato tipo pid_t
12    if (pid == 0)
13    {
14        printf("Soy el hijo %d mi padre es %d\n",getpid(),getppid());
15        printf("Me convierto en otra cosa\n"); //hasta aca, el hijo sigue siendo clon del padre
16        execl("./otracosa","./otracosa",0); //toma el binario del programa 'otracosa', ya no es clon
17        exit(7);
18    }
19    else
20        if (pid > 0)
21    {
22        printf("Soy el padre %d y mi hijo es %d y mi padre es %d\n",getpid(),pid,getppid());
23        printf("Padre dice: estoy esperando la terminacion del hijo\n");
24        hijoterm = wait(&status);
25        printf("Termino mi hijo %d\n",hijoterm);
26        //otra forma de mostrar el status
27        //printf("Mi hijo termino y me envio un %d\n",(status/256));
28        printf("Mi hijo termino y me envio un %d\n",WEXITSTATUS(status)); //para ver el valor de status
29    }
30    else // osea numero negativo
31        printf("fork fallo\n");
32    exit(6); // retorne la funcion principal (main)
33 }

```

En la línea 16 agregamos esta linea para el área de código donde reconocemos que es el hijo, para que cambie a ser el programa (“otracosa.c”). A partir de la línea 16, el proceso hijo deja de ser clon del proceso padre.



```

salo@DESKTOP-0M99BC2:~$ ./ejemplo-6.c
-bash: ./ejemplo-6.c: Permission denied
salo@DESKTOP-0M99BC2:~$ ./ejemplo-6
Soy el padre 382 y mi hijo es 383 y mi padre es 117
Soy el hijo 383 mi padre es 382
Padre dice: estoy esperando la terminacion del hijo
Me convierto en otra cosa
Soy el hijo convertido en otra cosa
Hola mundo cruel!
Termino mi hijo 383
Mi hijo termino y me envio un 0
salo@DESKTOP-0M99BC2:~$ 

```

Entonces, después que ejecute la línea ‘Me convierto en otra cosa’ el proceso hijo se convierte en el ejecutable correspondiente al código fuente (“otracosa.c”).

El programa hijo **NO** se tomó 7 segundos para terminar como dice en la línea 17 del código fuente, por que el **EXECL reemplazo el proceso hijo** a partir de esa misma línea, y las líneas 16 y 17 ya no están más en el proceso hijo, entonces el (“exit(7);”) ya no significa nada y por eso, al ejecutar el ejemplo-6, en la última línea dice “**Mi hijo termino y me envió un 0**” en vez de un 7 por que si nos fijamos en el código de fuente en la página 15 línea 8, dice (“**exit(0);**”).

Vamos a cambiar el argumento del **main** para que el proceso ejemplo-6 pueda recibir **string** (argumentos) desde la **Línea de comando**. Además, cambiamos la línea 16 para ver como funciona esto.

```

7 int main(int argc, char * argv[])
8 {
9     pid_t pid , hijoterm; // declarar una variable tipo pid_t
10    int status;
11    pid = fork(); // llamar a la funcion para que ejecute la llamada al sistema y retorna un dato tipo pid_t
12    if (pid == 0) //hijo
13    {
14        printf("Soy el hijo %d mi padre es %d\n"
15        "Me convierto en otra cosa\n");

```

```

7 int main(int argc, char * argv[])
8 {
9     pid_t pid , hijoterm; // declarar una variable tipo pid_t
10    int status;
11    pid = fork(); // llamar a la funcion para que ejecute la llamada al sistema y retorna un dato tipo pid_t
12    if (pid == 0) //hijo
13    {
14        printf("Soy el hijo %d mi padre es %d\n",getpid(),getppid());
15        printf("Me convierto en otra cosa\n"); //hasta aca, el hijo sigue siendo clon del padre
16        exec(argv[1],argv[1],0); //ya no es clon del padre
17        exit(7);
18    }

```

La idea es que el **ejemplo-6.c** reciba desde la **Línea de comando** un **string**, y ese string se lo pase por argumento al **EXECL**, esto quiere decir que si el string que paso desde la **Línea de comando** es el nombre de un ejecutable (por ej: "otracosa.c"), entonces **debería ejecutar ese código como proceso hijo**.

```

sal0@DESKTOP-0M99BC2:~$ ./ejemplo-6 ./otracosa
Soy el padre 483 y mi hijo es 484 y mi padre es 117
Soy el hijo 484 mi padre es 483
Padre dice: estoy esperando la terminacion del hijo
Me convierto en otra cosa
Soy el hijo convertido en otra cosa
Hola mundo cruel!
Termino mi hijo 484
Mi hijo termino y me envio un 0
sal0@DESKTOP-0M99BC2:~$
```

Ahora copie el código “otracosa.c” a un nuevo código llamado “otracosa1.c” para modificar algunas cosas, veamos el resultado al ejecutar el uno y después el otro:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     printf("Soy el hijo convertido en otra cosa 1\n");
7     printf("Hola mundo cruel 2020!\n");
8     exit(2);
9 }
```

```
salo@DESKTOP-0M99BC2:~$ cc otracosa1.c -o otracosa1
salo@DESKTOP-0M99BC2:~$ ./ejemplo-6 ./otracosa
Soy el padre 533 y mi hijo es 534 y mi padre es 117
Soy el hijo 534 mi padre es 533
Padre dice: estoy esperando la terminacion del hijo
Me convierto en otra cosa
Soy el hijo convertido en otra cosa
Hola mundo cruel!
Termino mi hijo 534
Mi hijo termino y me envio un 0
salo@DESKTOP-0M99BC2:~$ ./ejemplo-6 ./otracosa1
Soy el padre 535 y mi hijo es 536 y mi padre es 117
Soy el hijo 536 mi padre es 535
Padre dice: estoy esperando la terminacion del hijo
Me convierto en otra cosa
Soy el hijo convertido en otra cosa 1
Hola mundo cruel 2020!
Termino mi hijo 536
Mi hijo termino y me envio un 2
salo@DESKTOP-0M99BC2:~$
```

## C Linux Procesos 04

### Ponemos a trabajar al proceso padre y al proceso hijo

En este caso usaremos **open** para crear y abrir archivos, **write** para escribir en un archivo, **close** para cerrar un archivo y **sleep** para poner a dormir un proceso.

#### OPEN

##### NOMBRE

open , creat - abrir y posiblemente crear archivo

##### SYNOPSIS

```
#include <sys/types.h>
```

```
#include <fcntl.h >

int open ( const char * pathname , int flags ) ;
int open ( const char * pathname , int flags , mode_t mode ) ;

int creat ( const char * pathname , mode_t mode ) ;
.
```

## WRITE

### NOMBRE

write - escribir en el archivo

### SYNOPSIS

```
#include <unistd.h>
ssize_t write(int fd, const void*buf, size_t count);
```

## CLOSE

### NOMBRE

close - cerrar el archivo

### SYNOPSIS

```
#include <unistd.h>
int close(int fd)
```

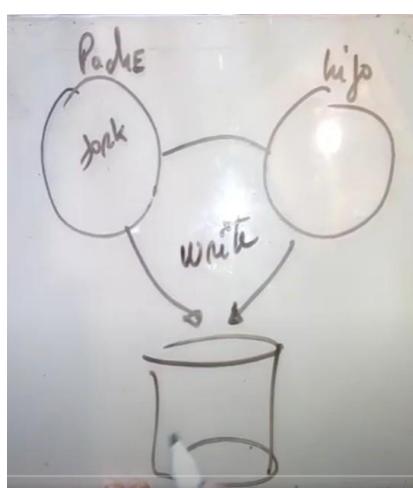
## SLEEP

### NOMBRE

sleep - dormir un proceso por el tiempo en específico

### SYNOPSIS

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```



El padre y el hijo van a escribir en un archivo en el file system. El archivo se va a llamar “abecedario” que va a tener las letras del alfabeto. Una letra escribe el padre, otra letra escribe el hijo, así hasta completar el abecedario. Para poder distinguir quién escribe que, el padre escribe en minúsculas y el hijo en mayúsculas.

La idea es entender la sincronización ENTRE procesos (entre dos procesos)

SLEEP no se puede usar para sincronizar (no lo vamos a usar)

Creamos un nuevo programa llamado “ejemplo0.c”

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     pid_t pid;
9     int fd = open("abecedario", O_CREAT|O_WRONLY|O_TRUNC, 0600);
```

open - la creación de un archivo con ‘open’

abecedario - nombre del archivo que va a tener el file system

O\_CREAT - flag de creación (es la letra o mayúscula, no cero)

O\_WRONLY - de solo escritura (es la letra o mayúscula, no cero)

O\_TRUNC - si tiene algo en el clon lo vamos a truncar (es la letra o mayúscula, no cero, puse = sin querer)

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     pid_t pid;
9     int fd = open("abecedario", O_CREAT|O_WRONLY|O_TRUNC, 0600); // 'fd' asi se conoce un
// archivo cuando un proceso esta en ejecucion
10    char letra = 'a';
11    pid = fork();
12    if (pid > 0) //padre
13    {
14        while(letra<='z') //mientras la letra sea menor que z
15        {
16            write(fd,&letra,1);
17            letra++; //nos crea la siguiente letra
18        }
19    }
20    else
21    if (pid == 0) //hijo
22    {
23        while(letra<='z') //mientras la letra sea menor que z
24        {
25            write(fd,&letra,1);
26            letra++; //nos crea la siguiente letra
27        }
28    }
29    else
30        perror("error fork\n");
31    close(fd);
32    return 0;
```

Así tendría que quedar el “ejemplo0.c”

**El write(fd,&letra,1); de la línea 16 significa:**

Escriba en el archivo (fd = file descriptor), el segundo argumento de write es (&letra) la dirección de memoria del buffer de salida (letra) por que la sintaxis de write nos pide casualmente su segundo argumento la dirección de memoria de un buffer, y la cantidad de byte que vamos a escribir (1) por que letras de char son 1 byte.

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo0.c -o ej0
salo@DESKTOP-0M99BC2:~$ ./ej0
salo@DESKTOP-0M99BC2:~$ ls -l
total 184
-rw----- 1 salo salo 52 Sep 14 17:37 abecedario
-rw-r--r-- 1 salo salo 141 Sep 11 00:10 ccejemplo.txt
-rwxr-xr-x 1 salo salo 16176 Sep 14 17:37 ej0
-rwxr-xr-x 1 salo salo 16000 Sep 11 10:13 ejemplo-1
-rw-r--r-- 1 salo salo 395 Sep 11 10:09 ejemplo-1.c
-rwxr-xr-x 1 salo salo 16000 Sep 11 14:35 ejemplo-2
-rw-r--r-- 1 salo salo 520 Sep 11 14:35 ejemplo-2.c
-rwxr-xr-x 1 salo salo 16128 Sep 11 15:07 ejemplo-3
-rw-r--r-- 1 salo salo 611 Sep 11 14:57 ejemplo-3.c
-rwxr-xr-x 1 salo salo 16304 Sep 11 17:30 ejemplo-4
-rw-r--r-- 1 salo salo 943 Sep 11 17:29 ejemplo-4.c
-rwxr-xr-x 1 salo salo 16248 Sep 11 17:52 ejemplo-5
-rw-r--r-- 1 salo salo 1163 Sep 11 17:52 ejemplo-5.c
-rwxr-xr-x 1 salo salo 16248 Sep 13 22:08 exemplo-6
-rw-r--r-- 1 salo salo 1261 Sep 13 21:40 exemplo-6.c
-rw-r--r-- 1 salo salo 126 Sep 11 00:16 exemplo.c
-rw-r--r-- 1 salo salo 725 Sep 14 17:33 ejemplo0.c
-rwxr-xr-x 1 salo salo 16000 Sep 13 20:13 otracosa
-rw-r--r-- 1 salo salo 157 Sep 13 20:12 otracosa.c
-rwxr-xr-x 1 salo salo 16000 Sep 13 22:13 otracosa1
-rw-r--r-- 1 salo salo 164 Sep 13 22:13 otracosa1.c
-rw-r--r-- 1 salo salo 141 Sep 11 00:10 texto.txt
salo@DESKTOP-0M99BC2:~$
```

Al compilar y ejecutar el ejemplo0.c , podemos ver en el directorio (comando "ls -l") que se creó un archivo llamado abecedario



Abrimos el gedit abecedario pero no se sabe que parte fue escrito por el padre y que parte por el hijo. Ahora vamos a modificar el código para ver quien escribe que.

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo0.c -o ej0
salo@DESKTOP-0M99BC2:~$ ./ej0
salo@DESKTOP-0M99BC2:~$ ls -l
total 184
-rw----- 1 salo salo 52 Sep 14 19:23 abecedario
```

Ejecutamos el **ejemplo0.c** que está como **ej0**, y chequeamos si está el archivo **abecedario**. Luego abrimos el **gedit** **abecedario**.



Lo que esta en **minuscula** es el **padre**, y lo que esta en **mayuscula** es el **hijo**

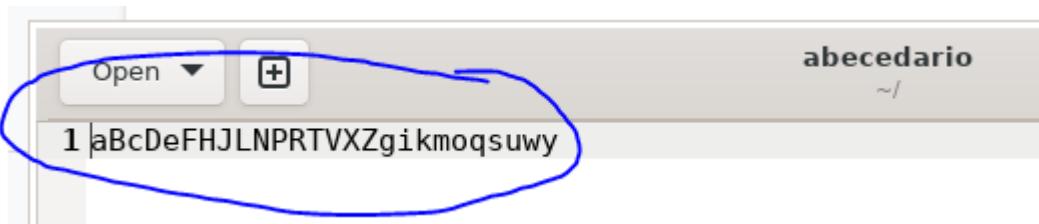
Nuestro objetivo es que se escriba el abecedario una sola vez, en vez de dos veces. Hace falta que el papa y el hijo cooperen para generar el abecedario en un archivo.

Copiamos el **ejemplo0.c** a uno nuevo (**ejemplo1.c**) y hacemos las siguientes modificaciones.

```
*ejemplo1.c
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     pid_t pid;
9     int fd = open("abecedario", O_CREAT|O_WRONLY|O_TRUNC, 0600); // 'fd' asi se conoce un
10    archivo cuando un proceso esta en ejecucion
11    char letra = 'a';
12    pid = fork();
13    if (pid > 0) //padre
14    {
15        while(letra <= 'z') //mientras la letra sea menor que z
16        {
17            write(fd,&letra,1);
18            letra+=2; //le suma 2 para que esten intercalado con el hijo (a,c,e,g...etc)
19        }
20    }
21    else
22    if (pid == 0) //hijo
23    {
24        letra = 'B';
25        while(letra <= 'Z') //mientras la letra sea menor que z
26        {
27            write(fd,&letra,1);
28            letra+=2; //le suma 2 para que esten intercalado con el padre (B,D,F,H...etc)
29        }
30    }
31    perror("error fork\n");
32    close(fd);
```

```
salo@DESKTOP-0M99BC2:~$ cc ejemplo1.c -o ej1
salo@DESKTOP-0M99BC2:~$ ./ej1
salo@DESKTOP-0M99BC2:~$ ls -l
total 220
-rwxr-xr-x 1 salo salo 16176 Sep 14 19:54 a.out
-rw----- 1 salo salo     26 Sep 14 19:54 abecedario
```

Abrimos el gedit abecedario después de ejecutar ej1

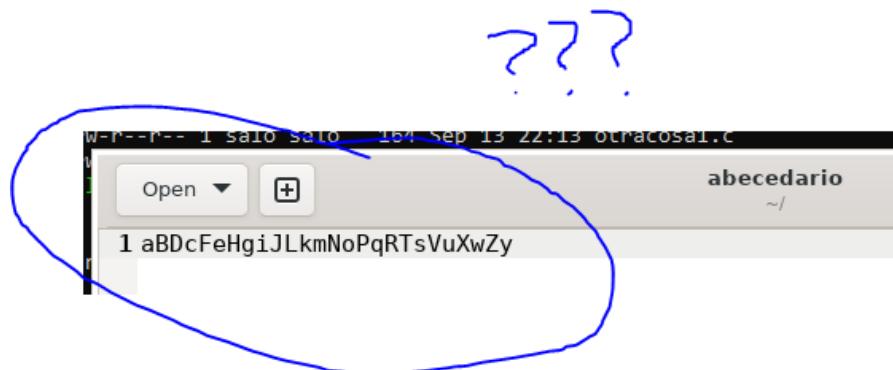


Logramos que cooperen el padre y el hijo a escribir el abecedario pero no está ordenado. Para lograr ordenarlos, aparece la segunda **sincronización**. (la primera **sincronización** que usamos era el **wait**).

```
6 int main(void)
7 {
8     pid_t pid;
9     int fd = open("abecedario", O_CREAT|O_WRONLY|O_TRUNC, 0600); // 'fd' asi se conoce un
    archivo cuando un proceso esta en ejecucion
10    char letra = 'a';
11    pid = fork();
12    if (pid > 0) //padre
13    {
14        while(letra <= 'z') //mientras la letra sea menor que z
15        {
16            write(fd,&letra,1);
17            letra+=2; //le suma 2 para que esten intercalado con el hijo (a,c,e,g...etc)
18            sleep(1); //concurrency, multiprogramacion
19        }
20    }
21    else
22    if (pid == 0) //hijo
23    {
24        letra = 'B';
25        while(letra <= 'Z') //mientras la letra sea menor que z
26        {
27            write(fd,&letra,1);
28            letra+=2; //le suma 2 para que esten intercalado con el padre (B,D,F,H...etc)
29            sleep(1); //concurrency, multiprogramacion
30        }
31    }
32    else
33    perror("error fork\n");
34    close(fd);
35    return 0;
```

El **sleep(1)** permite que le saquen el CPU al hijo luego de que ejecute su letra y se lo den al padre, después de que el padre ejecute una letra, duerme por 1 segundo y le pasan el CPU al hijo para que ejecute su letra, y así sucesivamente hasta llegar a la letra z.

Esta forma de usar el SLEEP para sincronización NO SE DEBE USAR NUNCA, pero a esta altura de nuestro aprendizaje todavía no aprendimos sobre los semáforos. Usamos esta estrategia solo para introducir este concepto de sincronización.



Por alguna razón el padre a veces viene primero, a veces segundo... (consultando con profesor para ver como arreglar esto)

## C Linux Procesos 04 Comentarios

Este video fue creado por el mismo problema que yo tuve con el abecedario.

A veces en algunas versiones de Linux en Debian y uBuntu, no siempre dejan ejecutar al padre primero, y luego el hijo.  
También depende de cuantos procesos están corriendo en el sistema, hay diferencia cuando uno tiene cientos de procesos y programas abiertos, y cuando uno tiene solo el ejemplo2.c corriendo.

En el video el profesor crea un programa que retorna 40 abecedarios para ver cual sale bien y cual sale mal. No hace falta hacer eso por que tiene muchas instrucciones que todavía no vimos, pero el punto es que entendamos la sincronización y que cada versión de linux es única.

### Conclusión del video

El riesgo de la sincronización con sleep:

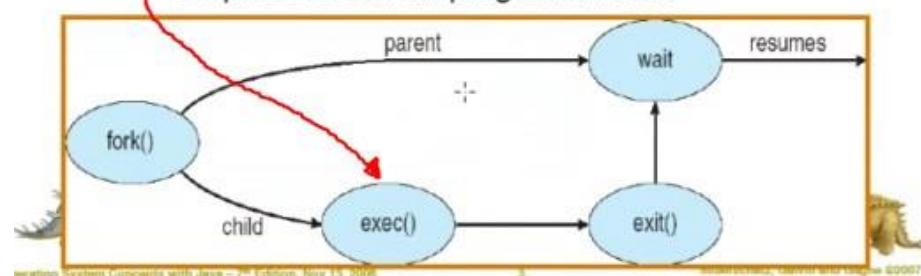
[Sección Crítica] cuando el padre está ejecutando la instrucción SLEEP, el padre puede despertarse antes de que el proceso hijo termine de ejecutar sus instrucciones y eso es exactamente lo que no se debe hacer y por qué no se debe usar SLEEP para sincronizar 2 procesos. Además el sleep nunca nos va a asegurar que durante el tiempo que ponemos en el argumento del sleep (sleep "( )") el hijo SIEMPRE va a terminar sus instrucciones dentro de ese tiempo. Para evitar esto, el argumento debe ser de un tiempo muy alto.

(por ej: `sleep(30)`) para el proceso padre, pero qué pasa si está ejecutando otra aplicación de mayor prioridad en el sistema, el sistema le va a quitar la atención del proceso hijo y se lo va a dar al otro proceso. Y así el proceso padre se despierta antes de que el proceso hijo ejecute sus instrucciones, luego de la instrucción `sleep` del padre, el hijo CONTINÚA con las instrucciones que le faltaba para terminar.

## Video Conferencia 02 - 28 / 8 / 2022

### Creación de procesos (Cont.)

- Espacio de direcciones
  - Hijo duplicado del padre
  - Se carga un programa en el hijo
- Ejemplos de llamadas al sistema en UNIX
  - `fork` para crear nuevos procesos
  - `exec`, después de `fork` reemplaza el espacio del proceso con un programa nuevo



Sabemos lo que hace el **EXEC** (mirar [página 15 a 18 de este documento](#))

### Creación de proceso en POSIX

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    /* fork: un proceso hijo */
    pid = fork();

    if (pid < 0) { /* Error */
        fprintf(stderr, "Falló el fork");
        exit(-1);
    } else if (pid == 0) { /* Proceso hijo */
        execvp("/bin/ls", "ls", NULL);
    } else { /* proceso padre */
        wait(NULL);
        printf("Concluyó hijo\n");
        exit(0);
    }
}
```

Este conjunto de código también entendemos lo que quiere mostrar (similar al código en la [página 12 de este documento](#)) excepto por la instrucción `execp`

A continuación está el código para copiar y pegar al editor y ver por nuestra propia cuenta como funciona en nuestra versión de linux comparado a lo del profesor a partir de **14:40**, **pasar a 21:43 para ver la ejecución.**

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void)
{
    pid_t pid;
    pid = fork();
    if (pid < 0) //Error
    {
        printf("Fallo el fork\n");
        exit(-1);
    }
    else if (pid == 0) //Proceso hijo
    {
        execlp("/bin/ls", "ls", "-l", NULL);
        printf("falló el execlp\n");
    }
    else //Proceso padre
    {
        wait(NULL);
        printf("Concluyó hijo\n");
        exit(0);
    }
}
```

Para crear directorio (carpeta) se usa el comando “ `mkdir` ”

```
salo@DESKTOP-0M99BC2:~/UAI-2022$ cc hijo01.c -o hijo01
salo@DESKTOP-0M99BC2:~/UAI-2022$ ./hijo01
total 36
-rwxr-xr-x 1 salo salo 16160 Sep 15 16:35 a.out
-rwxr-xr-x 1 salo salo 16120 Sep 15 16:42 hijo01
-rw-r--r-- 1 salo salo    501 Sep 15 16:41 hijo01.c
Concluyó hijo
salo@DESKTOP-0M99BC2:~/UAI-2022$
```

El hijo se convierte en el comando “ `LS -L` ”

Copiamos el archivo “hijo01.c” a uno nuevo “hijo02.c” y modificamos lo siguiente:

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6
7 int main(void)
8 {
9     printf("Yo soy el hijo convertido en otro codigo\n");
10    exit(0);
11 }
```

```
sal0@DESKTOP-0M99BC2:~/UAI-2022$ cc hijo02.c -o nuevo
sal0@DESKTOP-0M99BC2:~/UAI-2022$ ./nuevo
Yo soy el hijo convertido en otro codigo
sal0@DESKTOP-0M99BC2:~/UAI-2022$
```

Compilamos como “nuevo” y ejecutamos

Después utilizamos el comando “pwd” y copiamos la dirección de lo siguiente:

```
sal0@DESKTOP-0M99BC2:~/UAI-2022$ pwd
/home/salo/UAI-2022
```

Para pegarlo en el gedit de “hijo01.c” en lo siguiente:

```
Open ▾ + *hijo01.c
~/UAI-2022
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6
7 int main(void)
8 {
9     pid_t pid;
10    pid = fork();
11    if (pid < 0) //Error
12    {
13        printf("Fallo el fork\n");
14        exit(-1);
15    }
16    else if (pid == 0) //Proceso hijo
17    {
18        execl("/home/salo/UAI-2022/nuevo", "nuevo", NULL);
19        printf("falló el execl\n");
20    }
21    else //Proceso padre
22    {
23        wait(NULL);
24        printf("Concluyó hijo\n");
25        exit(0);
26    }
27 }
```

Al ejecutar, aparece así:

```
salo@DESKTOP-0M99BC2:~/UAI-2022$ cc hijo01.c -o hijo
salo@DESKTOP-0M99BC2:~/UAI-2022$ ./hijo
Yo soy el hijo convertido en otro codigo
Concluyo hijo
salo@DESKTOP-0M99BC2:~/UAI-2022$
```

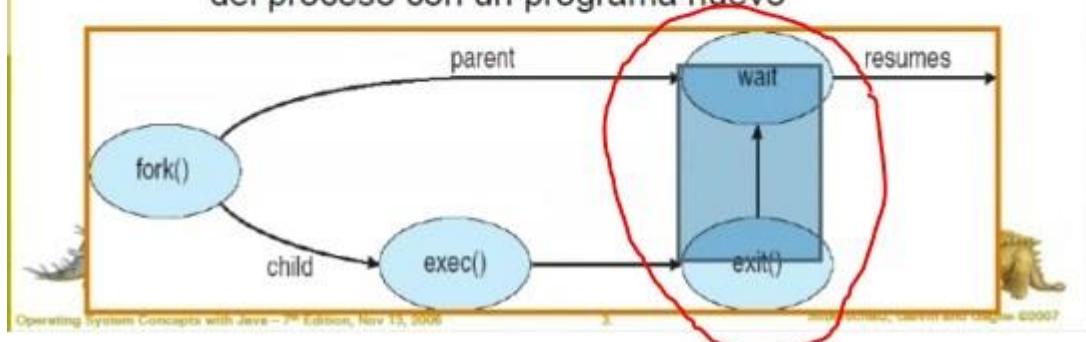
← EXEC

El programa “ nuevo ” lo abre el hijo desde el programa “ hijo01.c ” (basicamente lo mismo que hicimos en la página 14 hasta página 18 de este documento)

Después de eso, hasta 44:21 del VC muestra lo que hicimos en la página 11 y 12 de este documento para entender la relación entre PID y PPID.

## Creación de procesos (Cont.)

- Espacio de direcciones
  - Hijo duplicado del padre
  - Se carga un programa en el hijo
- Ejemplos de llamadas al sistema en UNIX
  - **fork** para crear nuevos procesos
  - **exec**, después de **fork** reemplaza el espacio del proceso con un programa nuevo



Esta sección entre el **exit** y el **wait** es lo más básico de una **sincronización** entre padre e **hijo** por que el **padre** espera que el **hijo** termine por que puede recibir un número entero del **hijo**.

Creamos un gedit con este programa.

```

1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include <sys/wait.h>
6
7 int main(void)
8 {
9     pid_t pid = fork();
10    if (pid < 0) //Error
11    {
12        printf("Fallo el fork\n");
13        exit(pid);
14    }
15    else if (pid == 0) //Proceso hijo
16    {
17        printf("yo soy el hijo mi pid es %d y el pid de mi padre es %d\n",getpid(),getppid());
18        system("ps -l"); //no es como execvp, esto literalmente transforma el hijo en vez de
 que el hijo abra una carpeta en especifico.
19        exit(7);
20    }
21    else //Proceso padre
22    {
23        int status = 0;
24        pid_t pidterm = wait(&status);
25        printf("yo soy su padre mi pid es %d y el pid de mi hijo es %d\n",getpid(),pid);
26        printf("termino mi hijo %d y eme envio un %d\n",pidterm, status/256);
27        exit(0);
28    }
29 }

```

```

salo@DESKTOP-0M99BC2:~/UAI-2022$ cc hijo02.c -o hijo02
salo@DESKTOP-0M99BC2:~/UAI-2022$ ./hijo02
yo soy el hijo mi pid es 943 y el pid de mi padre es 942
  F   S   UID      PID  PPID   C PRI  NI ADDR SZ WCHAN TTY          TIME CMD
  0 S  1000       11   10  0  80    0 -  4308 -      tty1    00:00:00 bash
  0 S  1000       90    1  0  80    0 -  4052 -      tty1    00:00:00 dbus-launch
  0 S  1000      942   11  0  80    0 -  2670 -      tty1    00:00:00 hijo02
  0 S  1000      943   942  0  80    0 -  2703 -      tty1    00:00:00 hijo02
  0 S  1000      944   943  0  80    0 -  2732 -      tty1    00:00:00 sh
  0 R  1000      945   944  0  80    0 -  4625 -      tty1    00:00:00 ps
yo soy su padre mi pid es 942 y el pid de mi hijo es 943
termino mi hijo 943 y eme envio un 7
salo@DESKTOP-0M99BC2:~/UAI-2022$

```

Acá vemos ese ejemplo de **sincronización muy básico**.

El profesor usa el comando “**echo \$?**” para ver el wait status del padre, osea muestra el **exit()** del padre, no del hijo. Ese número lo **recibe el padre del padre, osea shell (bash)** en nuestro caso.

En **1:10:35** dice estudiar hasta **página 24 (comunicación entre procesos) del PDF en ultra**.

## Itinerario 4 - Planificación de Procesos

### Niveles de planificación

- Planificación a **corto plazo** (Selecciona el siguiente proceso a ejecutar).
- Planificación a **medio plazo** (Selecciona qué procesos se añaden o se retiran (expulsión a swap) de memoria principal).
- Planificación a **largo plazo** (Realiza el control de admisión de procesos a ejecutar; Muy usada en sistemas batch).

### Tipos de planificación

- **No apropiativa**: El proceso en ejecución conserva el uso de la CPU mientras lo deseé.
- **Apropiativa**: El sistema operativo puede expulsar a un proceso de la CPU.

### Planificación: Medidas

- **Utilización de CPU**:
  - Porcentaje de tiempo que se usa la CPU.
  - Objetivo: Maximizar.
- **Productividad**:
  - Número de trabajos terminados por unidad de tiempo.
  - Objetivo: Maximizar.
- **Tiempo de retorno ( $T_q$ )**
  - Tiempo que está un proceso en el sistema. Instante final ( $T_f$ ) menos instante inicial ( $T_i$ ).
  - Objetivo: Minimizar.
- **Tiempo de servicio ( $T_s$ )**:
  - Tiempo dedicado a tareas productivas (cpu, entrada/ salida).  $T_s = T_{CPU} + TE/S$
- **Tiempo de espera ( $T_e$ )**:

– Tiempo que un proceso pasa en colas de espera.  $T_e = T_q - T_s$

- *Tiempo de retorno normalizado ( $T_n$ ):*

– Razón entre tiempo de retorno y tiempo de servicio.  $T_n = T_q/T_s$

– Indica el retardo experimentado.

### Ejemplos de Planificaciones:

**First to Come First to Serve:** Primer en llegar primero en servir

**Shortest Job First:** Primero el trabajo más corto.

Cíclico o Round-Robin

Asignación por prioridades

Múltiples colas con retroalimentación.

## Planificación FCFS

Un planificador FCFS (**First-Come, First-Served**) asigna la CPU a los procesos que están en estado listo según su orden de llegada.

### Planificación First-Come, First-Served (FCFS)

Proceso	Burst time
P <sub>1</sub>	24
P <sub>2</sub>	3
P <sub>3</sub>	3

Lo que está en rojo muestra la orden de llegada de los procesos, 1 siendo primero y 3 tercero

Lo que está en azul es el tiempo que ese proceso va a usar la CPU



- Tiempo de espera para  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Tiempo promedio de espera:  $(0 + 24 + 27)/3 = 17$

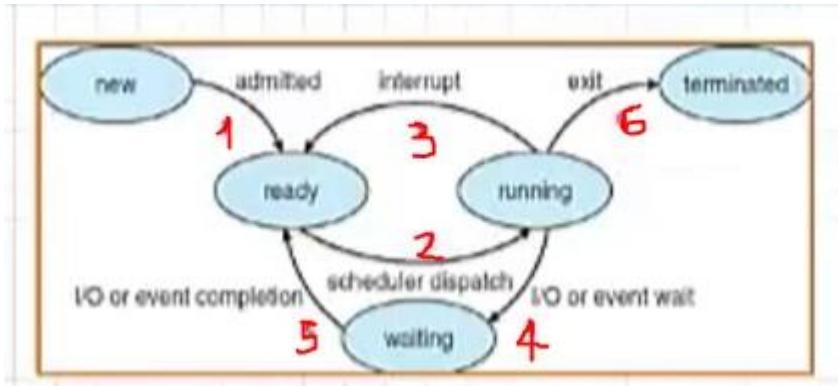
**Overhead (sobrecarga)** - El tiempo que va a utilizar la CPU del sistema operativo para administrar la planificación de todos los procesos.

Proceso	Características																	
PA	CPU 40 ms, E/S disco 30 ms, CPU 10 ms, E/S cinta 20 ms, CPU 10 ms																	
PB	CPU 40 ms, E/S cinta 40 ms, CPU 30 ms, E/S disco 30 ms, CPU 10 ms																	

Las interrupciones, incluido el overhead duran 10 mseg.

El sistema cuenta con dos dispositivos de E/S, uno para unidad de cinta y otro para disco.

Tiempo	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250	260	270	280	290	300	310	320	330
SO	E	E	E				E	E		E			E	E	E	E	E	E	E	E	E			E	E	E	E	E	E				
Proceso A				E	E	E	E	D	D	D						E	C	C							E								
Proceso B								E	E	S	E	E	C	C	C	C		S	E	E	E	D	D	D			E						
Interrupcion																																	
Rutina PA	1		2						4			5					4			5				2		6							
Rutina PB		1								2				4				5	2				4			5	2		6				
Cola de Listos		A	A	B	B	B	B	B	B			A	A	A	A				B	A	A	A	A	A			B		0				
		B																															



Explicación del Excel con relación al diagrama de flujo (por cada 10 ms) :

(en 23:48 del video de Youtube: PlanificaciónFCFS del itinerario 4 - planificaciones de proceso)

**10 ms** : El SO realiza una rutina 1 para el **proceso A** para cambiar el estado de nuevo a listo.

**20 ms** : El SO realiza una rutina 1 para el **proceso B** para cambiar el estado de nuevo a listo.

**30 ms** : El SO realiza una rutina 2 para el primer proceso que llegó a listo (**proceso A**) y le da la CPU para que ejecute su primer rafaga de CPU (total de **40 ms** como dice en el primer cuadro de características) osea desde **30 ms** hasta **70 ms**.

**80 ms** : el so atiende la interrupción y realiza una rutina 4 para el **proceso A**.  
*(interrupciones son de 10 ms)*

**90 ms** : La CPU está libre para asignarse a otro proceso, el SO realiza una rutina 2 para el **proceso B**.

**100 ms** : El **proceso A** está en una espera de **entrada / salida (E/S)** y el **proceso B** comienza a ejecutar su primera rafaga de CPU (tiene que ser de **40 ms**).

**120 ms** : el so recibe la interrupción del control del disco y entonces suspende (s) el **proceso B** para atender la interrupción del hardware, entonces realiza una rutina 5 para el **proceso A**.

**130 ms** : Continua la ejecución del **proceso B**.

**150 ms** : se realiza una interrupción, el so atiende la llamada al sistema y realiza la rutina 4 para el **proceso B**.

**160 ms** : El SO está libre y realiza una rutina 2 para el **proceso A** y pone a **proceso B** en espera de **entrada / salida (E/S)**.

**170 ms** : El **proceso A** ejecuta su rafaga de CPU y el **proceso B** sigue en espera de **entrada / salida (E/S)**

**180 ms** : hay llamada de interrupción del **proceso A**, el so atiende la llamada del sistema y realiza una rutina 4 para el **proceso A**.

**190 ms** : Tenemos el **proceso B** que tiene la cinta y el **proceso A** que no puede tener la cinta hasta que la cinta sea liberada, pero aquí podemos considerar que el **proceso A** está en espera.

**200 ms** : hay interrupción que atiende el SO para realizar una rutina 5 para el **proceso b**.

**210 ms** : el so realiza rutina 2 para el **proceso B** y el **proceso A** empieza a realizar la **entrada / salida (e/s)** sobre la cinta.

**220 ms** : el so atiende interrupción y hace una rutina 5 para el **proceso A**.

**230 ms** : deja la cpu ejecutar el **proceso B** que ya lo tenía.

**260 ms** : hay llamada de interrupción del **proceso B**, el so atiende la llamada y hace una rutina 4 para el **proceso B** pasar a estado de espera.

**270 ms** : la cpu fue asignada a otro proceso, entonces el so hace una rutina 2 para el **proceso A**.

**280 ms** : el **proceso A** hace una llamada al sistema por que quiere terminar (exit).

**290 ms** : el so atiende la interrupción y hace una rutina 6 para el **proceso A** después de su rafaga de ejecución. el **proceso B** está esperando que se resuelva la **entrada / salida (e/s)** sobre el disco.

**300 ms** : el so atiende la interrupción por hardware y atiende la rutina 5 para el **proceso B**.

**310 ms** : la cpu queda libre para asignarse a otro proceso, entonces el so hace una rutina 2 para el **proceso B**.

**320 ms** : realiza la última ráfaga de ejecución del **proceso B** y hace una llamada al sistema por que finaliza.

**330 ms** : el so atiende la llamada para finalizar el proceso con una rutina 6 para el **proceso B** por que quiere terminar (exit).

Analizando el excel que muestra la ejecución de los 2 procesos:

**Porcentaje de uso de CPU** - La cantidad de tiempo que la CPU estuvo en funcionamiento (todas las letras E) dividido el tiempo total de uso de la CPU (estado activo) y como vemos en el milisegundo 190 el CPU no estuvo en funcionamiento.

$$320 \text{ ms} / 330 \text{ ms} = 0,97 \rightarrow 97\%$$

**Tiempo ocioso de CPU** - Calculamos el porcentaje que el CPU no estuvo activo (en ms 190).

$$10 \text{ ms} / 330 \text{ ms} = 0,03 \rightarrow 3\%$$

**Productividad** - **Proceso A** y **Proceso B** (2 procesos) en **330 ms**

**Sobrecarga del SO** - El tiempo que el sistema operativo usa la CPU para planificar (tiempo de administración). Todos los E en la línea de SO del excel.

$$180 \text{ ms} / 330 \text{ ms} = 0,54 \rightarrow 54\%$$

**Tiempo medio de retorno** - La suma del tiempo que terminó el proceso A más la suma del tiempo que terminó el proceso B.

$$( 290 + 330 ) / 2 = 310 \text{ ms}$$

**Tiempo medio de espera** - La suma del tiempo que estuvo el **proceso A** en estado listo más la sumatoria del tiempo que estuvo el **proceso B** en listo dividido la cantidad de procesos. Sumar los A en la cola de listos y los B en la cola de listos, luego dividir por dos.

$$( \text{110} + \text{90} ) / 2 = \text{100 ms}$$

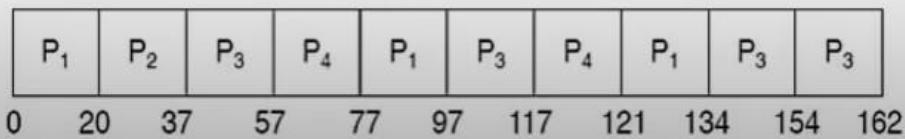
## Planificación RR

### Planificación Cíclico o Round-Robin

#### Ejemplo de RR con Time Quantum = 20

Proceso	Burst Time
$P_1$	$53 - 20 = 33$
$P_2$	$17 - 17 = \text{Exit}$
$P_3$	$68 - 20 = 48$
$P_4$	$24 - 20 = 4$

- La gráfica de Gantt es:



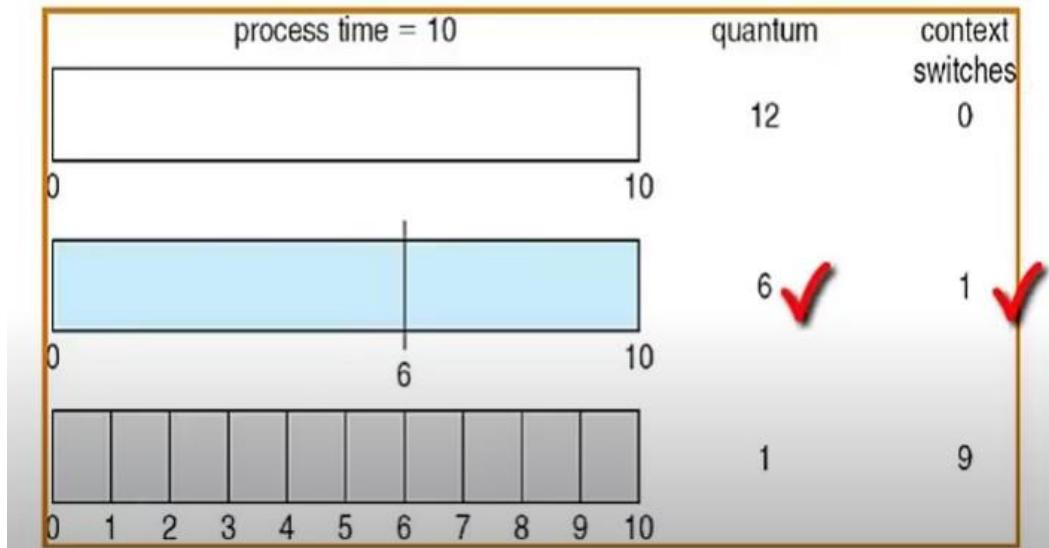
Básicamente lo que hace el **Round Robin con Time Quantum = 20** es que le da 20 unidades de tiempo a cada proceso hasta que sean terminados.

Como vemos en el ejemplo, **proceso 1** tiene una rafaga de 53 milisegundos, osea que se va a ejecutar por 20 milisegundos y le va a dar el CPU al **proceso 2** para que ejecute durante los siguientes 20 milisegundos que le proporciona pero como el proceso 2 solo dura 17 milisegundos en ejecutar por completo y terminar, a los 17 milisegundos se pasa la CPU al **proceso 3**, y luego al **proceso 4**, así sucesivamente hasta que termine todos los procesos.

## Tiempos de Quantum y Context Switch

El tiempo de **Quantum** puede variar y se relaciona con el **context switch**. Como el ejemplo de abajo muestra, si el Quantum time es de 1 milisegundo el tiempo de proceso es de 10 milisegundos, habrá 9 context switches.

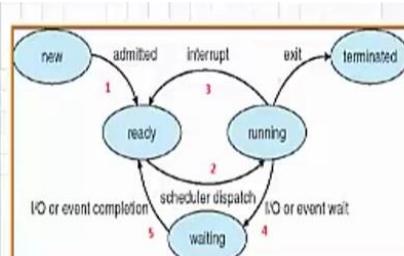
### Tiempo de Quantum y Context Switch



En minuto 5:20 el profesor realiza el mismo ejercicio de planificación del proceso A y proceso B que está en la página 33 de este documento pero aplicando la planificación Round Robin con un Quantum de 20 milisegundos.

Consideré un sistema de multiprogramación que debe ejecutar dos procesos con las siguientes características y usando una planificación de CPU RR

Proceso	Características
PA	CPU 40 ms, E/S disco 30 ms, CPU 10 ms, E/S cinta 20 ms, CPU 10 ms
PB	CPU 40 ms, E/S cinta 40 ms, CPU 30 ms, E/S disco 30 ms, CPU 10 ms



Porcentaje de uso de CPU	= $380 / 420 = 0,90 = 90\%$
Tiempo ocioso de CPU	= 10 %
Productividad	= 2 procesos en 420 maes
Sobre carga del SO	= $240 / 420 = 0,57 = 57\%$
Tiempo Medio de Retorno	= $ 320 + 420  / 2 = 370 ms$
Tiempo Medio de Espera	= $(120 + 150) / 2 = 135 ms$

## Planificación Por Prioridades Expulsiva

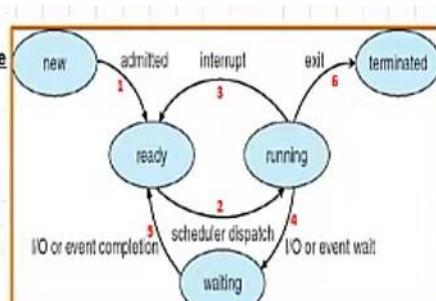
Una prioridad numérica se asocia con cada proceso.

El CPU se asigna al proceso con la prioridad más alta (**entero más pequeño = prioridad más alta**)

- Preemptive (rutina 3 y rutina 5)
  - Nonpreemptive (rutina 1 y rutina 4)

A partir de 6:21 el profesor explica como va completando el ejercicio de Excel aplicando la planificación de CPU por prioridades con desalojo o apropiativa. Aca explica mejor lo que yo hice en este documento en página 34 y 35 con lo de 'ms'.

Considere un sistema monoprocesamiento con multiprogramación concurrente que debe ejecutar dos procesos con las siguientes definiciones, usando una planificación de CPU por prioridades con desalojo o apropiativa.



Porcentaje de uso de CPU	= $360/390 = 92,31\%$
Porcentaje de tiempo ocioso de CPU	= $30/390 = 7,69\%$
Productividad	= 2 procesos en 390 ms
Porcentaje de tiempo de sobrecarga del SO	= $210/390 = 53,83\%$
Tiempo Medio de Retorno	= $(390+270)/2 = 330$
Tiempo Medio de Espera	= $(170+60)/2 = 115$

## Video Conferencia 03 - 04 / 9 / 2022

### Despachador

El módulo despachador da el control de CPU al proceso seleccionado por el planificador:

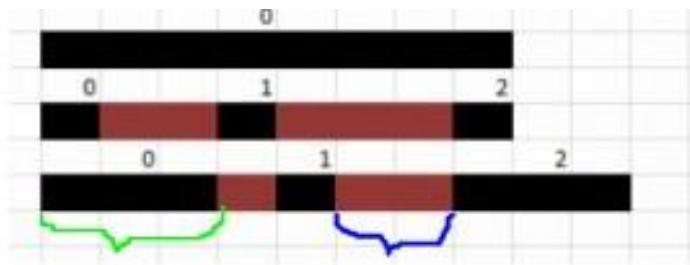
- Cambiar el contexto
- Cambiar a modo de usuario
- Brincar a la posición adecuada en el programa de usuario para reiniciar el programa

Latencia de despacho — tiempo que toma al despachador detener un proceso e iniciar otro

### Criterios de planificación

## Criterios de planificación

- Utilización de CPU – mantener el CPU tan ocupado como sea posible
- Rendimiento (Throughput) – # de procesos que completan su ejecución por unidad de tiempo
- Tiempo de vuelta – cantidad de tiempo para ejecutar un proceso particular
- Tiempo de espera – cantidad de tiempo que un proceso ha esperado en la cola listos
- Tiempo de respuesta – cantidad de tiempo que toma desde que una solicitud se realiza hasta que se produce la primera respuesta, no salida (para ambientes de tiempo-compartido)



EJECUTANDO      E/S

Aca solo muestra que un proceso puede tener **secciones de E/S (entrada salida)** y **secciones de rafaga de CPU**.

Como vemos, el primer proceso (la barra todo negro) tiene una rafaga larga de CPU sin secciones de entrada/salida. Y los otros tienen rafagas de CPU y rafagas de E/S.

Ahora, el planificador **no toma en cuenta las rafagas de E/S**, osea estos existen en el proceso pero el **planificador solo planifica rafagas de CPU** (sección negra de las barras).

### Planificación FCFS (Cont.)

- Si los procesos llegan en el orden:  $P_2 | P_3 | P_1$
- La gráfica de Gantt para el planificador es:



- Tiempo de espera para  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Tiempo promedio de respuesta:  $(6 + 0 + 3)/3 = 3$
- Mucho mejor que en el caso anterior
- **Efecto Convoy** procesos cortos detrás de procesos largos

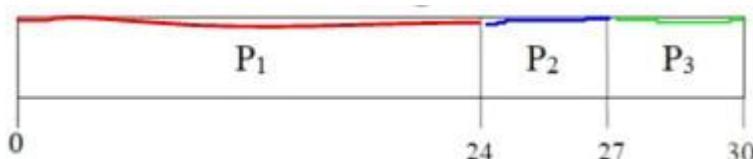
Suponemos que tenemos 3 procesos con estos tiempos de rafagas de CPU suponiendo que no tienen rafagas de E/S.

**P1 : 24 milisegundos**

**P2 : 3 milisegundos**

**P3 : 3 milisegundos**

La gráfica de Gantt del planificador de **first-come first-served (FCFS)** dice que el primero que llega, usa el CPU primero. Entonces, como ejemplo usando los 3 procesos: la gráfica quedaría así:



El tiempo de espera para el **P1** es **0 ms**, por que entro primero entonces está usando el CPU para que ejecute inmediatamente y usa el CPU durante los **24 ms** que necesita el **P1**.

El tiempo de espera para el **P2** es **24 ms**, por que tiene que esperar que el **P1** termine su ejecución. Cuando El SO asigna el CPU al **P2**, lo va a usar durante los **3 ms** que necesita el proceso.

El tiempo de espera para el **P3** es **27 ms**, por que tiene que esperar que el **P1** y **P2** terminen su uso del CPU.

El tiempo promedio de **espera** sería: **(0 ms + 24 ms + 27 ms) / 3 = 17 ms**

Ahora supongamos que los procesos llegan en diferente orden:

**P2 : 3 milisegundos**

**P3 : 3 milisegundos**

**P1 : 24 milisegundos**

La gráfica de Gantt del planificador de **first-come first-served (FCFS)** con el orden de esta manera quedaría así:



Entonces el tiempo promedio de **espera** sería: **(0 ms + 3 ms + 6 ms) / 3 = 3 ms**

El **Efecto Convoy** dice que si primero empiezan los procesos de **rafaga** de CPU más corto, el tiempo de espera promedio se disminuye.

## Planificación Shortest-Job-First (SJF)

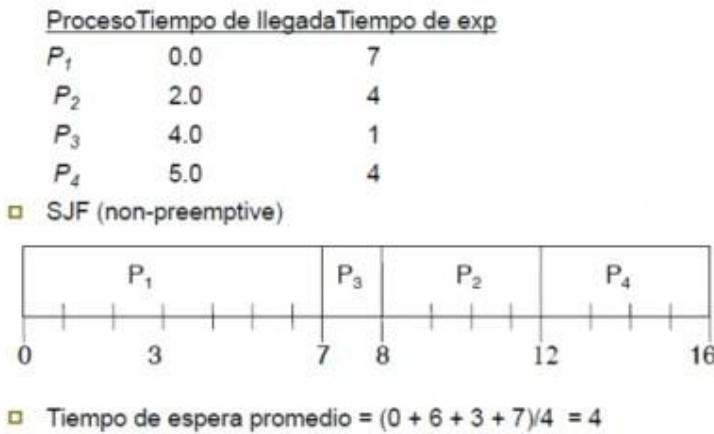
- Asociar a cada proceso la **longitud de su siguiente explosión** de CPU. Planificador selecciona el de tiempo más corto
- **Dos esquemas:**
  - **nonpreemptive** – una vez que le damos el CPU a un proceso dado, no puede quitársele hasta que complete su explosión de CPU
  - **preemptive** – si un proceso nuevo llega al CPU con longitud de explosión menor al tiempo restante del proceso en ejecución, lo sacas. Este esquema es conocido como Shortest-Remaining-Time-First (SRTF)
- **SJF es optimo** – da el **menor tiempo de espera promedio** para un conjunto de procesos dado

## Planificación Shortest-Job-First (SJF) (rafaga más corto primero)

El algoritmo viene en dos esquemas:

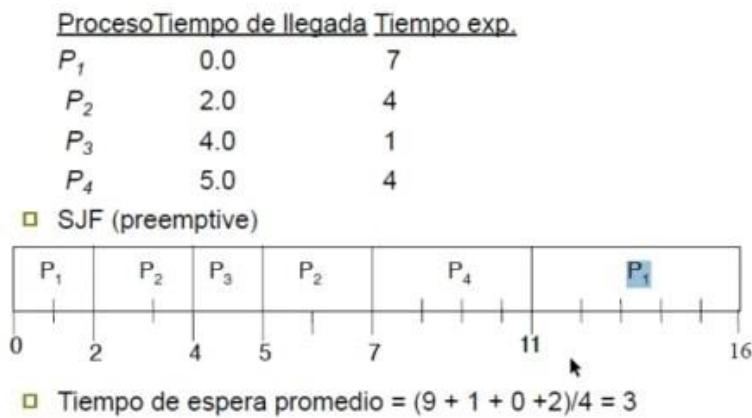
- **nonpreemptive:** (no apropiativo) este se lo llama **SJF**
- **preemptive:** (apropiativo) este se lo llama **SRTF (Shortest-Remaining-Time-First)**

### Ejemplo de Non-Preemptive SJF



El profesor explica este ejemplo en VC 03 tiempo 1:02:26

### Ejemplo de Preemptive SJF



El profesor explica este ejemplo en VC 03 tiempo 1:06:02

## Video Conferencia 04 (1/2) - 11 / 9 / 2022

Ecuación de como determina el sistema para determinar el tiempo de la próxima rafaga de CPU en proceso:

$$S_{n+1} = wT_n + (1 - w)S_n$$



Lo que está en rojo es la **rafaga de CPU** (puede ser de 3 ms el primero y de 2 ms el segundo)

Lo que está en azul es 2 ms de **entrada / salida** (discontinuación del uso del CPU)

**S<sub>0</sub>**: necesitamos calcular el pronóstico para la rafaga 0 (primer 3 ms).

**S<sub>1</sub>**: después de que S<sub>0</sub> ejecute, necesitamos calcular el pronóstico para la rafaga S<sub>1</sub>.

**n+1**: Esto me está diciendo el número de rafaga del cual estamos haciendo el pronóstico.  
Entonces...

**S<sub>n+1</sub>**: es el cálculo del pronóstico para la siguiente rafaga del CPU.

**T<sub>n</sub>**: duración real de la última rafaga ejecutada.

Entonces para este ejemplo...

Si quiero calcular **S<sub>1</sub>** (segunda rafaga): **S<sub>1</sub> = T<sub>n</sub>** y **T<sub>n</sub>** es igual a **3 ms** porque la rafaga de **S<sub>0</sub>** es de **3 ms**.

Nuestra ecuación queda así: **S<sub>1</sub> = w3 + (1 - w) x 3**

El **S<sub>n</sub>** es el valor de pronóstico que se ha ejecutado (**S<sub>0</sub> de 3 ms**), porque S<sub>n+1</sub> es para calcular el siguiente pronóstico, S<sub>n</sub> es el valor del último ejecutado.

El '**w**' es un valor de peso, y ese valor SIEMPRE está **0 <= w <= 1** (entre 0 y 1)

Si **w = 0**

**S<sub>n+1</sub> = S<sub>n</sub>**

(El pronóstico para la siguiente rafaga (**S<sub>n+1</sub>**) es igual al pronóstico del anterior (**S<sub>n</sub>**)).

Tomando el ejemplo sería: S<sub>n+1</sub> = 0 x 3 + (1 - 0) x 3 → **S<sub>n+1</sub> = 3 ms**

Si **w = 1**

**S<sub>n+1</sub> = T<sub>n</sub>**

(El pronóstico para la siguiente rafaga (**S<sub>n+1</sub>**) es igual a la duración real de la rafaga anterior (**T<sub>n</sub>**)).

Tomando el ejemplo sería: S<sub>n+1</sub> = 1 x 3 + (1 - 1) x 3 → **S<sub>n+1</sub> = 3 ms**

Si **w = ½**

**S<sub>n+1</sub> = ½T<sub>n</sub> + ½S<sub>n</sub>**

(50% de peso a la duración real y 50% de peso al pronóstico anterior)

Ahora si **w = 0**, todas las rafagas van a ser **igual**.

si **w = 1**, todas las rafagas van a ser **diferente**.

Pero, si **w = ½** podemos **estimar un promedio** para la siguiente rafaga.

Pero hay otro problema, como calculo el pronóstico para la **PRIMERA rafaga** de CPU (**S<sub>n+1</sub>**) ?

$n+1 = 0 \rightarrow n = -1$  ... y no existe posición negativa de rafaga porque la primera rafaga es la rafaga 0

Entonces lo que quiere decir  $n+1 = 0 \rightarrow n = -1$  es que **NO se puede** calcular el pronóstico de la primera rafaga.

## Planificación con prioridades

```
salo@DESKTOP-0M99BC2:~$ ps -l
F S   UID    PID  PPID C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000     11    10  0  80    0 -  4308 -        tty1      00:00:00 bash
0 R  1000     78    11  0  80    0 -  4625 -        tty1      00:00:00 ps
salo@DESKTOP-0M99BC2:~$
```

Los procesos tienen prioridad (**PRI**) asignada por el sistema.

```
salo@DESKTOP-0M99BC2:~$ ps -lax
F   UID    PID  PPID PRI  NI   VSZ   RSS WCHAN STAT TTY          TIME COMMAND
0     0      1    0  20    0  8948  400 ?       Ssl ?      0:00 /init
0     0     10    1  20    0  9296  228 -       Ss  tty1      0:00 /init
0  1000    11    10  20    0 17232 3856 -       S  tty1      0:00 -bash
0  1000    79    11  20    0 18500 1836 -       R  tty1      0:00 ps -lax
salo@DESKTOP-0M99BC2:~$
```

Ese valor de prioridad es el que va tener en cuenta el algoritmo de **planificación con prioridad**.

El CPU se asigna al proceso con la prioridad **más alta** (**entero más pequeño**).

Igual que el algoritmo SJF ([página 42 de este documento](#)), puede venir en forma:

- **Preemptive** (apropiativa (STRF & Prioridad))
- **Nonpreemptive** (no apropiativa (FCFS & SJF & Prioridad))

El **SJF** es un **planificador con prioridades** donde la prioridad es la **duración de la siguiente rafaga de CPU**.

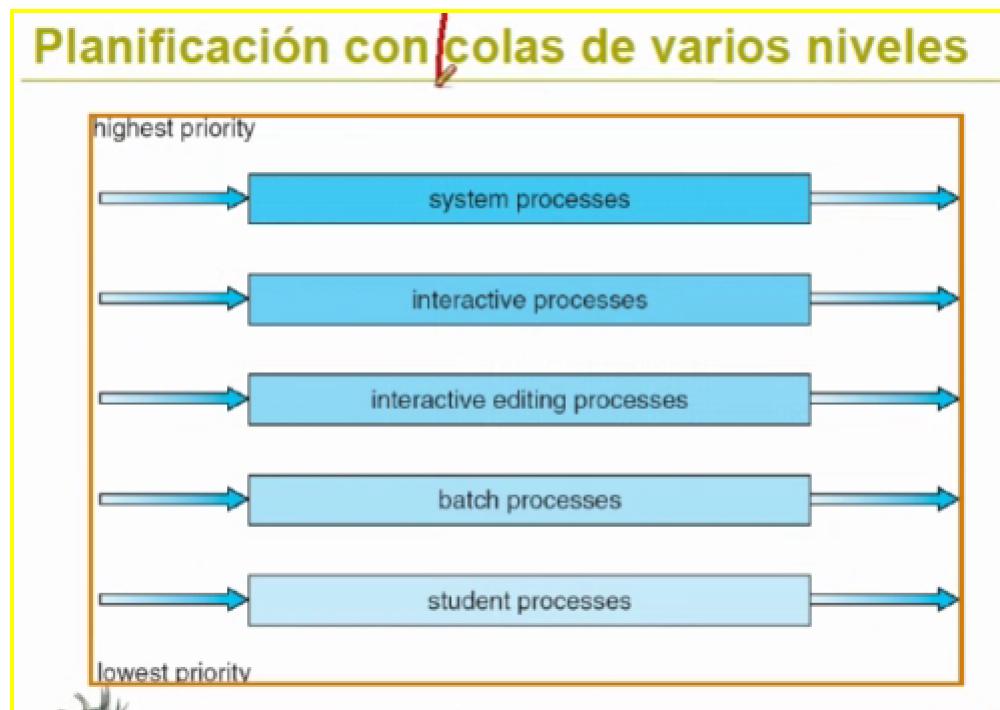
- **Hambruna**: procesos de baja prioridad pueden nunca ejecutarse [[problema](#)]
- **Envejecer**: conforme pasa el tiempo, incrementar la prioridad de los procesos [[solucion](#)]

### Repaso de planificación Round Robin

## Round Robin (RR)

- Cada proceso obtiene una pequeña rebanada de tiempo de CPU (*time quantum*), usualmente 10-100ms.
  - Acabado el tiempo, el proceso es sacado (preempted) y se agrega a la cola de listos.
- Sea  $n$  el número de procesos y  $q$ , la rebanada de tiempo.
  - Cada proceso obtiene  $1/n$  del tiempo de CPU en rebanadas de, a lo más,  $q$  unidades a la vez.
  - Ningún proceso espera más de  $(n-1)q$  unidades.
- Rendimiento
  - $q$  grande  $\rightarrow$  FIFO
  - $q$  pequeña  $\rightarrow q$  debe ser grande con respecto al cambio de contexto, de otra forma la carga administrativa es muy grande

### Colas multi - niveles



Se arma colas de procesos con más prioridad a procesos con menos prioridad:

### Alta prioridad

- Procesos del sistema
- Procesos de interacción
- Procesos de edición
- Procesos en ejecución batch
- Procesos estudiantes

### Baja prioridad

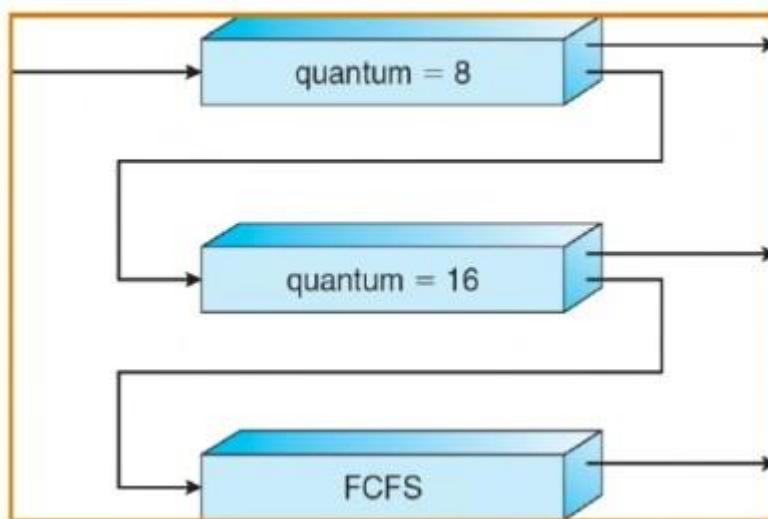
En esta planificación, primero se **termina de ejecutar todos los procesos de sistema (alta prioridad)**, y después empieza con la segunda cola de procesos, y así hasta llegar a la cola de menor prioridad.

### Colas varios niveles: parámetros

#### Colas varios niveles

- Envejecimiento: proceso se mueve entre colas
- Planificador para colas de varios niveles, parámetros:
  - Número de colas
  - **Algoritmos de planificación para cada cola**
  - Método utilizado para determinar **cuando avanzar un proceso**
  - Método utilizado para determinar **cuando retrasar un proceso**
  - Método utilizado para determinar **a qué cola entra un proceso cuando requiera servicio**

#### Colas con varios niveles y retroalimentación



Mirar a partir de **30:55** para ver la explicación de este gráfico.

**Video Conferencia 04 (2/2) - 11 / 9 / 2022**

## Ejercicios de planificación

3.7 Considere un sistema de multiprogramación que debe ejecutar dos procesos con las siguientes características:

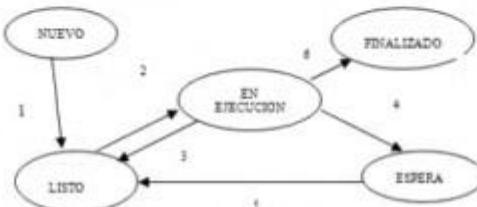
Proceso	Características
A	CPU 40 ms, E/S disco 30 ms, CPU 10 ms, E/S cinta 20 ms, CPU 10 ms
B	CPU 40 ms, E/S cinta 40 ms, CPU 30 ms, E/S disco 30 ms, CPU 10 ms

**Considerar:**

- Las interrupciones, incluido el overhead duran 10 mseg.
  - El sistema cuenta con dos dispositivos de E/S, uno para unidad de cinta y otro para disco.

Se pide completar el diagrama que esta a continuación para los siguientes algoritmos:  
a) ESES

- a) FCFS
  - b) SJF sin desalojo
  - c) SRTF con desalojo
  - d) Round Robin con quantum de 30 ms.



**Mirar a partir de 7:42 para ver la solución del ejercicio.**

(EXCEL FCFS) (EXCEL STRF) (EXCEL SJF) (EXCEL PRIO B APROPIATIVO)  
(EXCEL RRQ30)

Rendimiento de CPU (porcentaje de uso de la CPU) =  $320 / 330 = 0.9666 \rightarrow 97\%$

Tiempo medio de retorno = **(TBA + TRB) / cantidad de procesos** = **(290 + 300) / 2 = 295**

Productividad = cantidad de procesos, tiempo total = 2 procesos, 330 ms

$$\text{Tiempo medio de espera} = \frac{\text{TEA} + \text{TEB}}{\text{cantidad de procesos}} = \frac{(110 + 90)}{2} = 100 \text{ ms}$$

Sobre carga o latencia (overhead) = Tiempo que trabajó el SO = **180** / 330 = **54%**

Tenemos los datos del planificador en función de los criterios de planificación FCFS. Ahora para practicar, hacer este mismo ejercicio pero con planificación SJF, SRTF, round robin y planificación por prioridades ('apropiativa' y otro con 'no apropiativa')

## Itinerario 5: Hilos y Procesos

Los hilos se consideran una unidad básica de utilización de la CPU y cada uno comprende:

- Identificador de thread
- Contador de programa
- Conjunto de registros
- Pila

Comparten con el resto de hilos del proceso:

- Mapa de memoria (sección de código, sección de datos, shmem)
- Ficheros abiertos
- Señales, semáforos y temporizadores

### YouTube 01: Sistemas Operativos, Hilos 1 Concepto

- Concepto
- Beneficios de los hilos
- Hilos a nivel kernel y a nivel usuario
- Modelos multihilos
- Arquiteturas Multicore
- Hilos Posix

#### Concepto

Un hilo es una línea de ejecución de un proceso.

```
int main()
{
    int i;
    printf("Inicio\n");

    for(i=0;i<2;i++)
        printf("Hola %d\n",i);

    printf("Fin");
}
```

Un hilo es una línea de ejecución de un proceso. Osea que si vemos la secuencia de instrucciones que se está ejecutando en la foto y pudiéramos trazar una línea por todas las instrucciones que se están ejecutando, se podría verlo como un hilo así:

```

int main()
{
    int i;
    printf("Inicio\n");

    for(i=0;i<2;i++)
        printf("Hola %d\n",i);

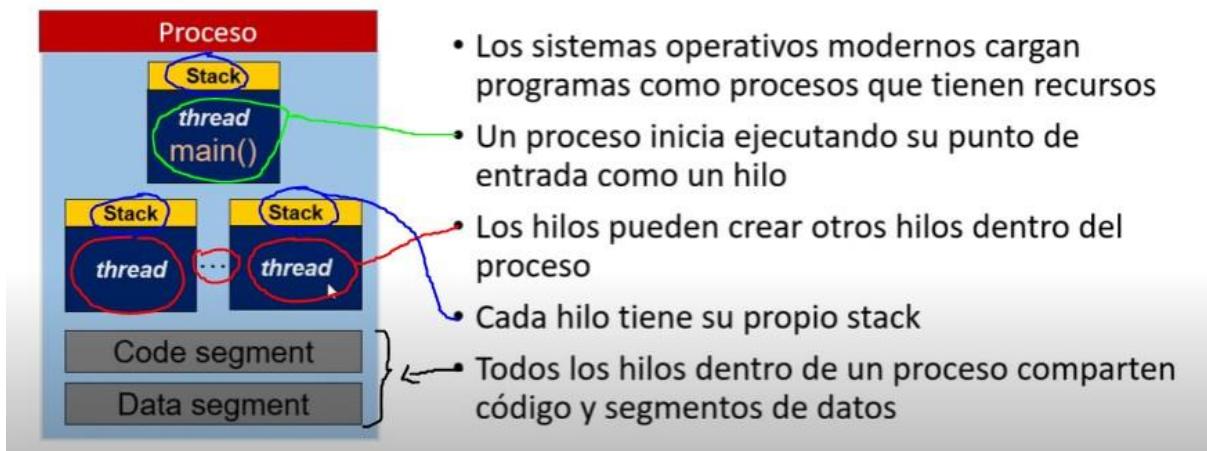
    printf("Fin");
}

```

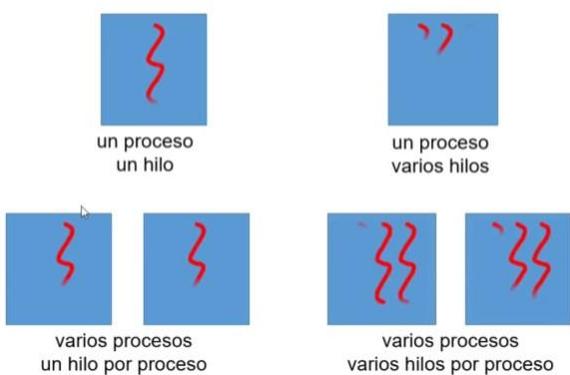
Todo proceso parte inicialmente con un **único hilo principal**. Pero el sistema operativo ofrece llamadas al sistema que permiten al programador crear y destruir hilos. De esta manera, un proceso está **compuesto por uno o más hilos**.

### Procesos y sus recursos

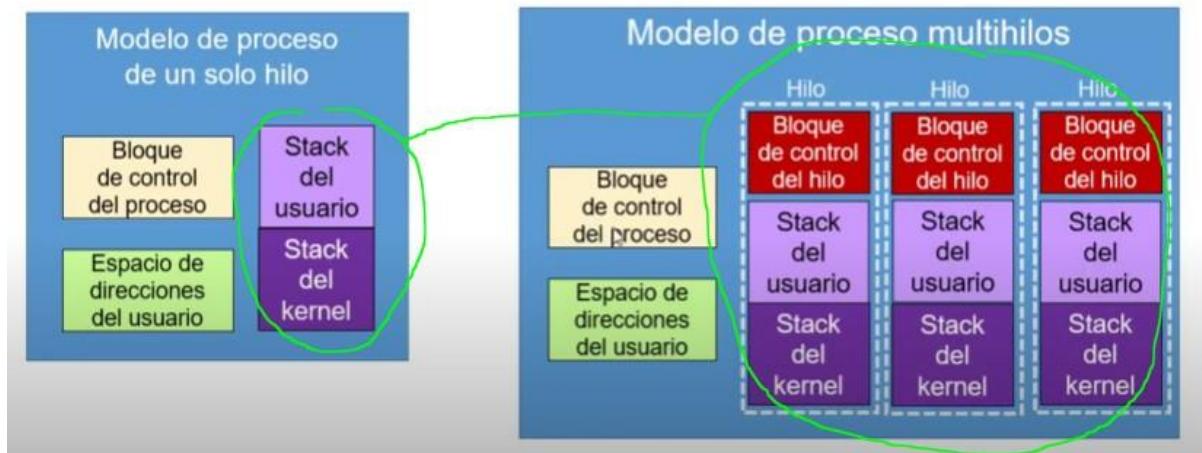
## Procesos e Hilos



## Procesos e hilos



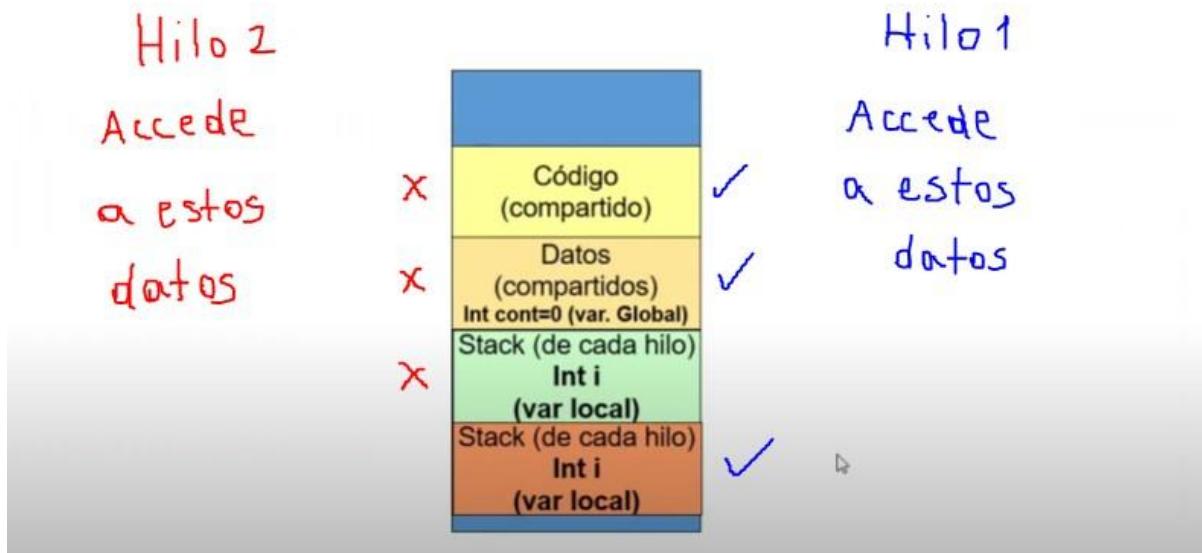
### Modelos de procesos de un solo hilo y de muchos hilos



En **multihilos**, por cada hilo va a tener un:

- bloque de control de hilo.
- stack de user.
- stack de kernel.

## Hilos en memoria



Los hilos **comparten segmento de código y datos**, pero **cada hilo debe tener su stack**.

### Sistema Operativo multihilo

- Sistema operativo que mantiene varios hilos de ejecución dentro de un mismo proceso.
- MS - DOS soporta un solo proceso con un solo hilo.
- UNIX soporta múltiples procesos de usuario, pero solo un hilo por proceso.
- Windows, Solaris, Linux, OS X, y OS/2 soportan múltiples hilos.

## YouTube 02: Sistemas Operativos, Hilos 2 Beneficios de los hilos

### Beneficios de los hilos



Con un solo hilo, una aplicación se **ejecuta secuencialmente** aunque utilice diferentes recursos sin dependencias, por ej: puede ejecutar una sección de código sin depender que la anterior haya sido terminada.

### Multihilo

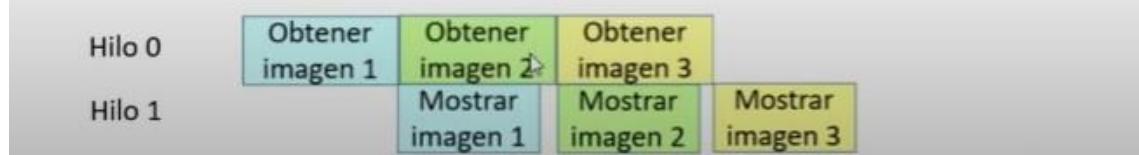


Con varios hilos un **hilo puede estar haciendo procesamiento** (uso del CPU) mientras otro **hilo efectúa operaciones de E/S**.

Un ejemplo es: Un navegador web puede tener **un hilo para leer las imágenes** de la red, mientras que **otro hilo las está mostrando** en pantalla.



### Multihilo



Acá vemos que un **multihilo acelera el proceso** de mostrar y obtener imágenes.

Otro ejemplo es: Un procesador de texto puede tener un hilo para **mostrar gráficos**, otro hilo para responder al **tecleo del usuario** y un tercer hilo para realizar la **revisión ortográfica y gramatical** en segundo plano.

### Multihilo



Entonces, las aplicaciones también se pueden diseñar para aprovechar las capacidades de procesamiento en sistemas multicore. Dichas aplicaciones pueden realizar varias tareas intensivas de procesamiento en paralelo en múltiples cores.

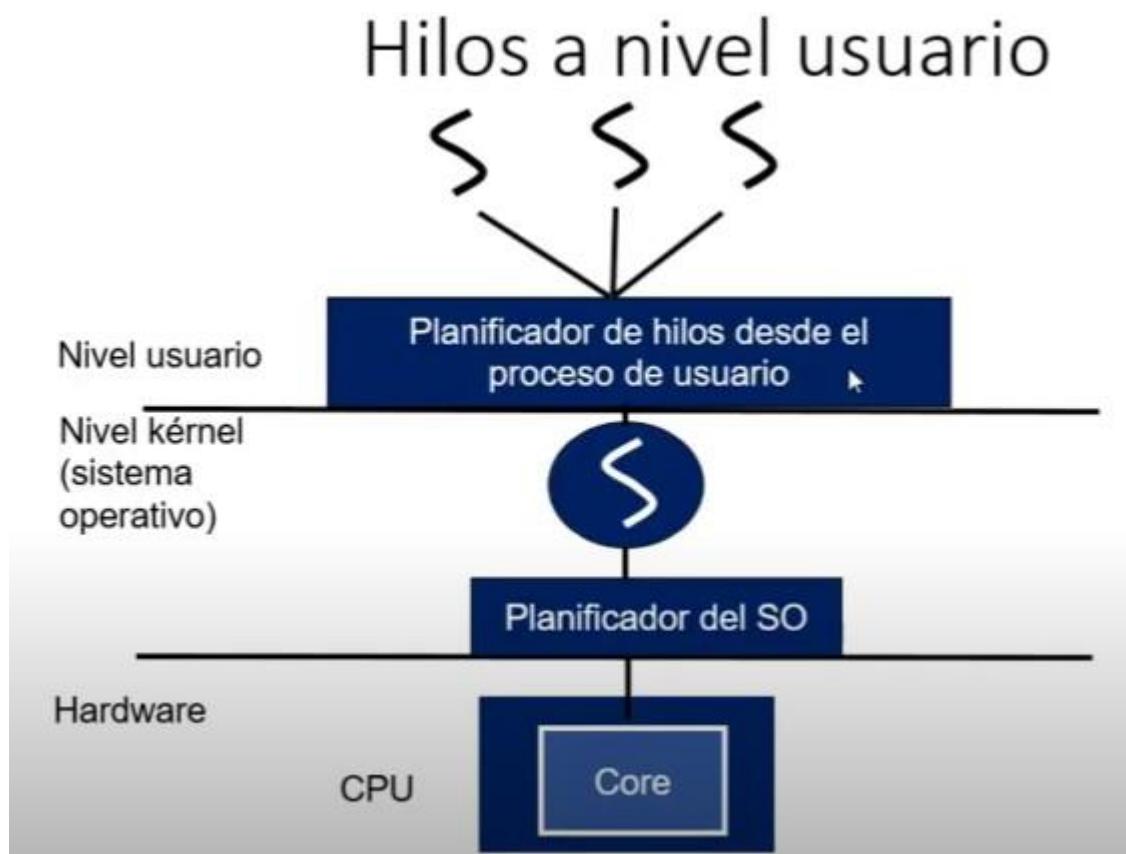
Por ejemplo: Minería de datos, gráficos, inteligencia artificial, ordenamiento, etc.

¿Qué pasa si tenemos 4 CPU y solo 1 hilo? No vamos a estar aprovechando los otros 3 CPU porque 1 hilo requiere 1 CPU nomas.



Pero al dividir el trabajo entre los 4 CPU el proceso se ejecuta mucho más rápido.

[YouTube 03: Sistemas Operativos, Hilos 3 Hilos a nivel kernel y a nivel usuario](#)



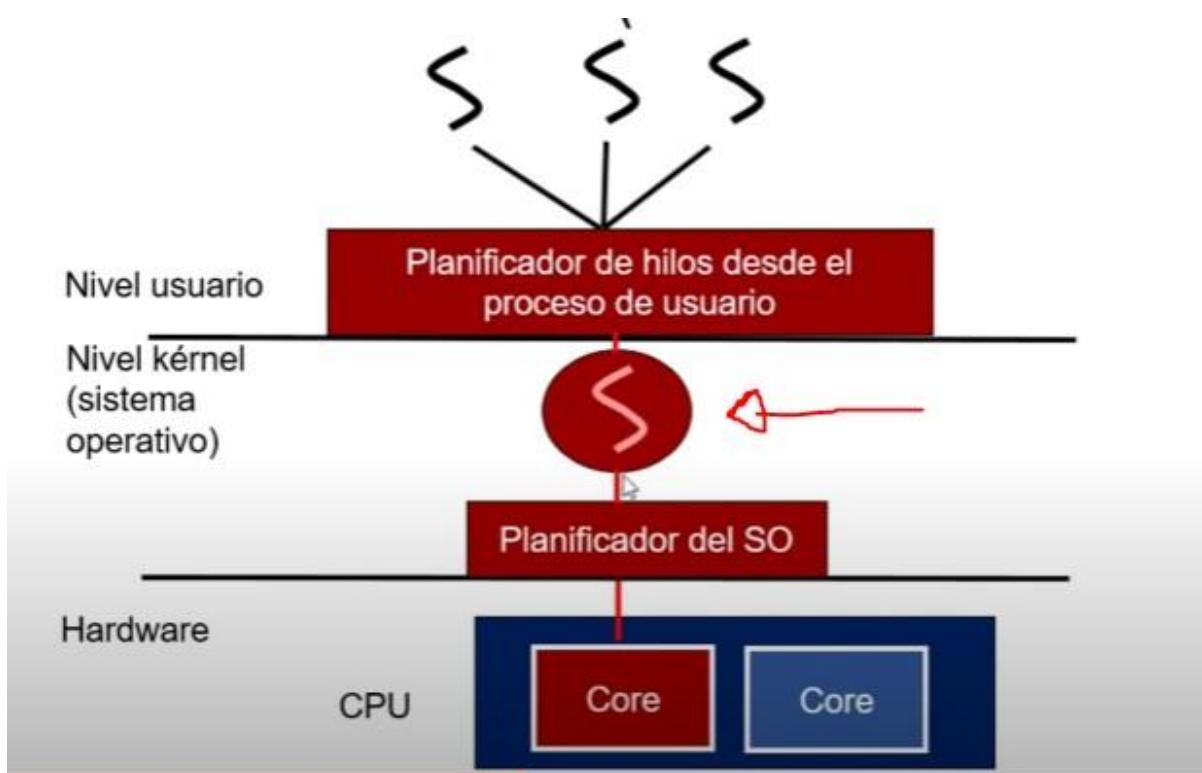
Los hilos van a existir solamente en el espacio de usuario, es decir, van a estar en el programa del usuario pero los **hilos no van a ser vistos** por el **sistema operativo**.

Por lo tanto, a nivel usuario, tenemos un **planificador de hilos** y este manejo de hilos lo hace una **librería de hilos**. El **planificador puede seleccionar cual de los hilos del programa es el que va a estar en ejecución**.

*Los hilos podían implementarse en las aplicaciones que se ejecutaban en sistemas operativos que no son capaces de planificar hilos*

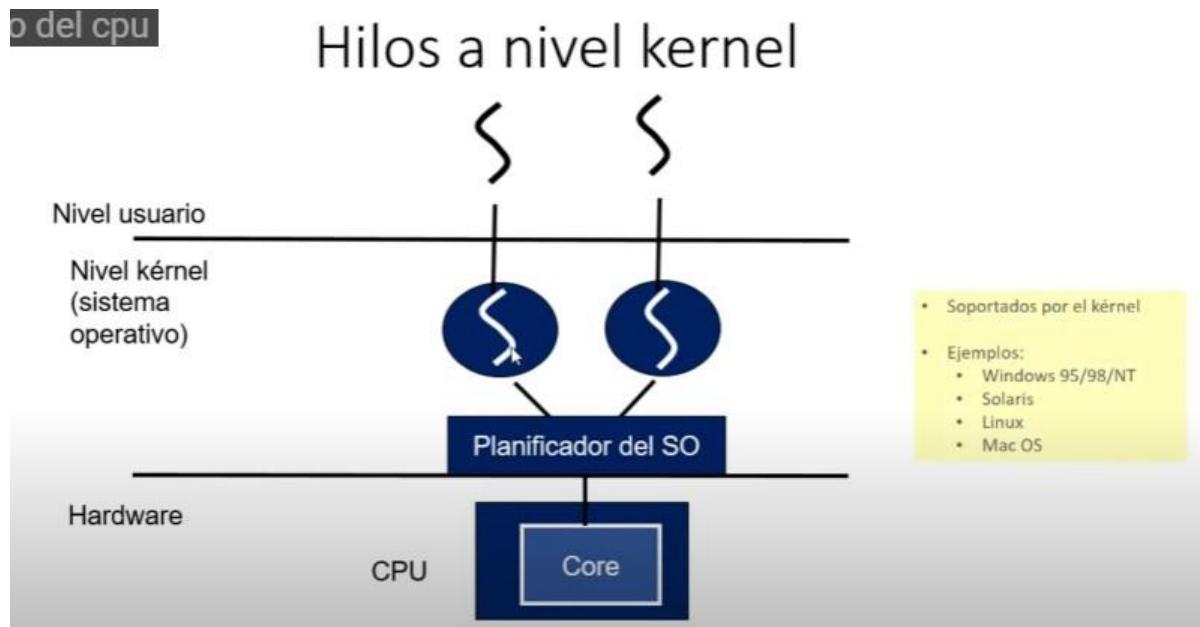
Ejemplo: Primeras implementaciones de UNIX

**Hilos a nivel usuario pero en una arquitectura donde hay un CPU con dos o más núcleos:**



Como solamente va a haber un hilo a nivel kernel, el planificador del sistema operativo solamente va a ser **capaz de planificar ese hilo en uno de los núcleos**. Entonces, **aunque el programa tenga muchos hilos**, estos hilos van a ser planificados desde el proceso de usuario, pero a nivel del kernel solamente va a existir un hilo que se va a **ejecutar en uno de los núcleos**.

Hilos a nivel kernel en una arquitectura donde hay un CPU con un núcleo:

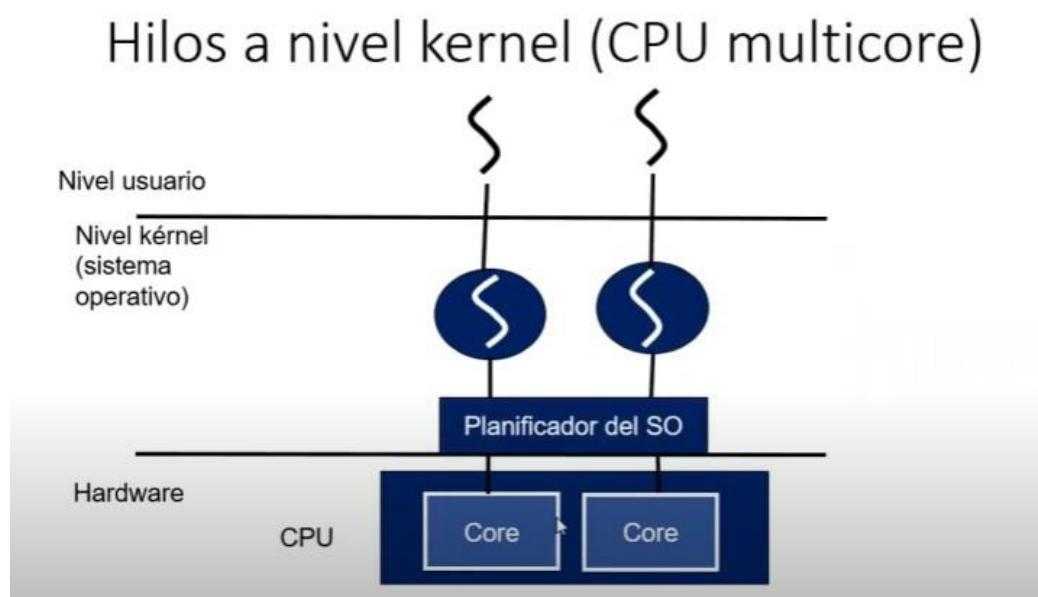


A nivel kernel, ahora los hilos existen en el sistema operativo. El planificador del sistema operativo es el que se va a encargar de decidir cuál de los hilos es el que va tener el tiempo del CPU.

Prácticamente todos los **sistemas operativos modernos** usan este sistema de hilos soportados a nivel kernel.

El proceso de usuario no se tiene que encargar de la planificación de los hilos

Hilos a nivel kernel en una arquitectura donde hay un CPU con multi núcleos:



Ahora que sucede con los hilos a nivel kernel cuando hay un CPU con multiple núcleos? El planificador del sistema operativo puede asignar cada hilo con un núcleo para que se ejecuten en paralelo y cada uno de los hilos pueden estar haciendo tareas diferentes dentro del mismo proceso.

#### YouTube 04: Sistemas Operativos, Hilos 4 Modelos multihilos

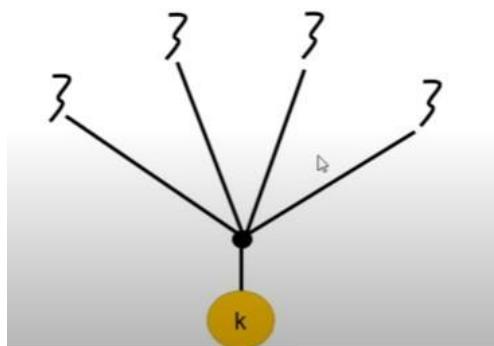
##### Modelos multihilos

Estos son la relación que puede haber entre **hilos a nivel usuario** y **hilos a nivel kernel**.

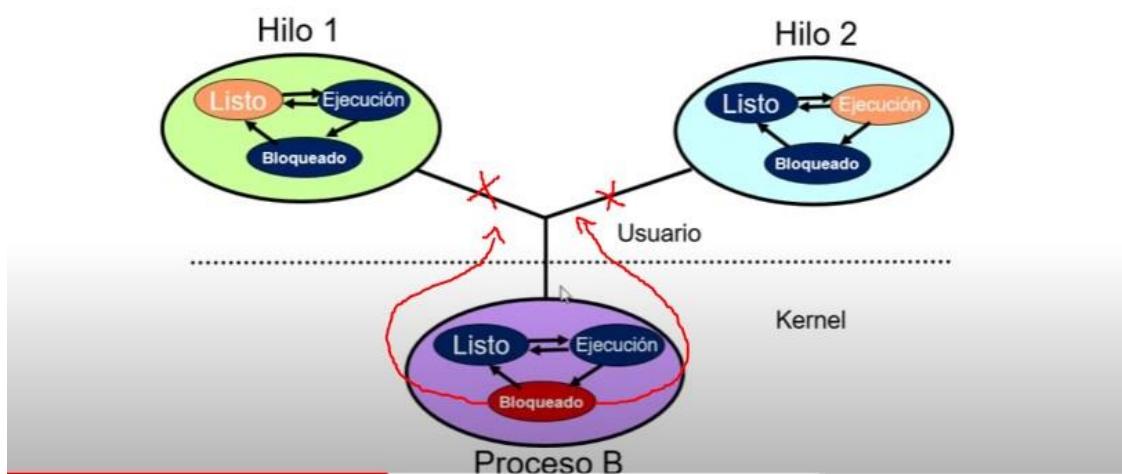
- **Muchos a uno**
- **Uno a uno**
- **Muchos a muchos**

##### Muchos a uno

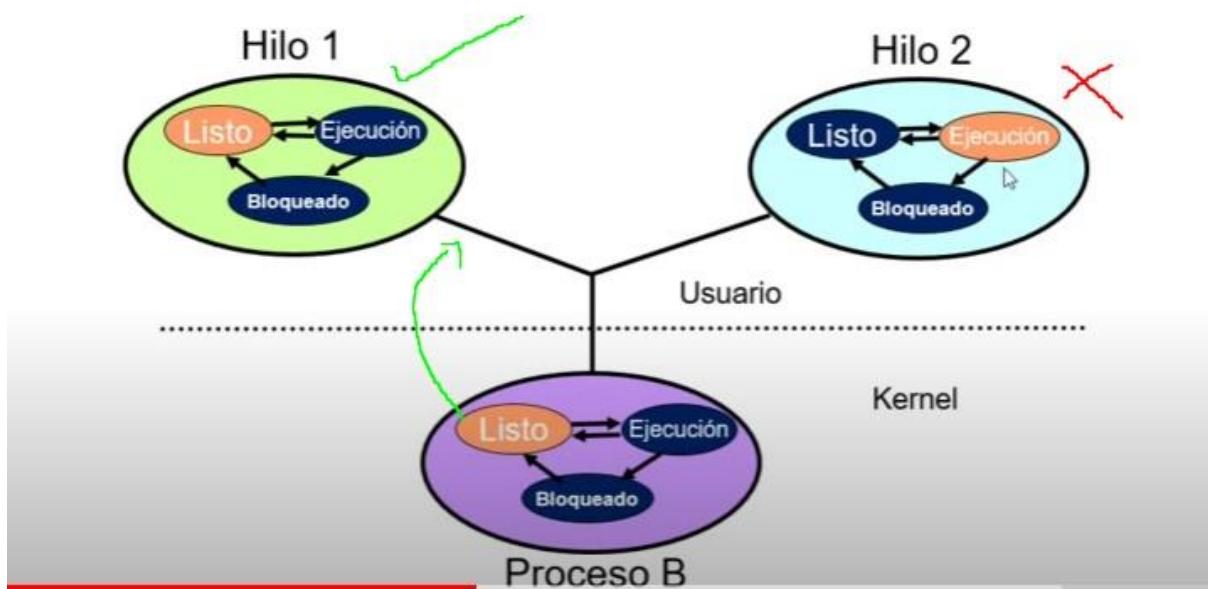
- Muchos hilos a **nivel usuario** mapeados a un hilo a **nivel kernel**.
- Usado en sistemas que no soportan **hilos** a nivel kernel.



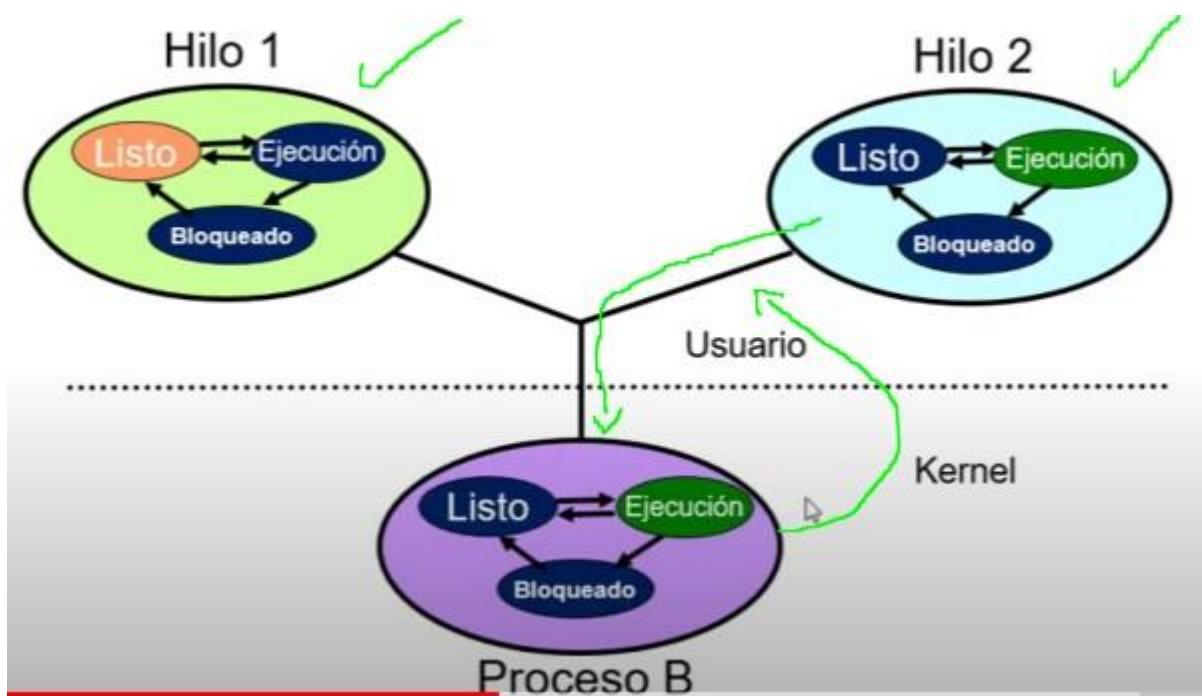
##### Relaciones entre estados de ULTs y estados de procesos



Acá vemos que si el **proceso está bloqueado**, todos los hilos a nivel usuario van a estar **bloqueados** aunque, como vemos en el ejemplo, el hilo 2 está en ejecución, no va a estar ejecutando por que el hilo a nivel kernel (o el proceso) está **bloqueado**.



En este ejemplo vemos que el hilo a nivel kernel (proceso) está en estado **listo** y aunque hilo 2 a nivel usuario está en ejecución, no se va a ejecutar.



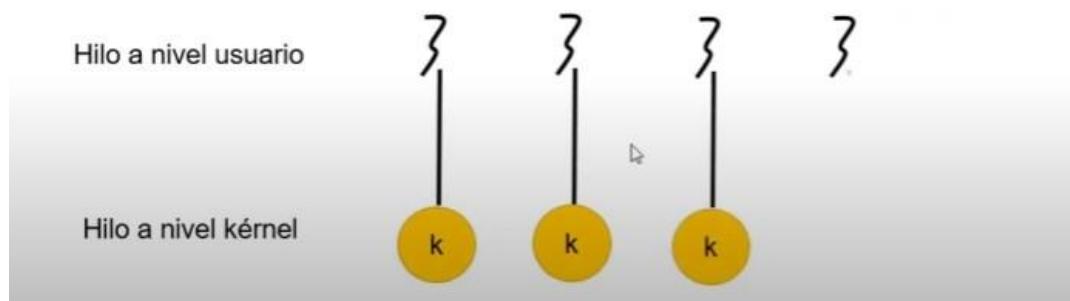
Ahora cuando el proceso pasa a **ejecución** (por que el sistema operativo decide que tiene que pasar a ejecución) el hilo 2 va a estar en **ejecución**.

Y si de repente entra el planificador a nivel usuario y pone al hilo 2 en estado listo y pasa el hilo 1 a estado ejecución, el hilo 1 es el que se va a ejecutar, **solamente mientras el hilo a nivel kernel esté en estado de ejecución.**

### Uno a uno

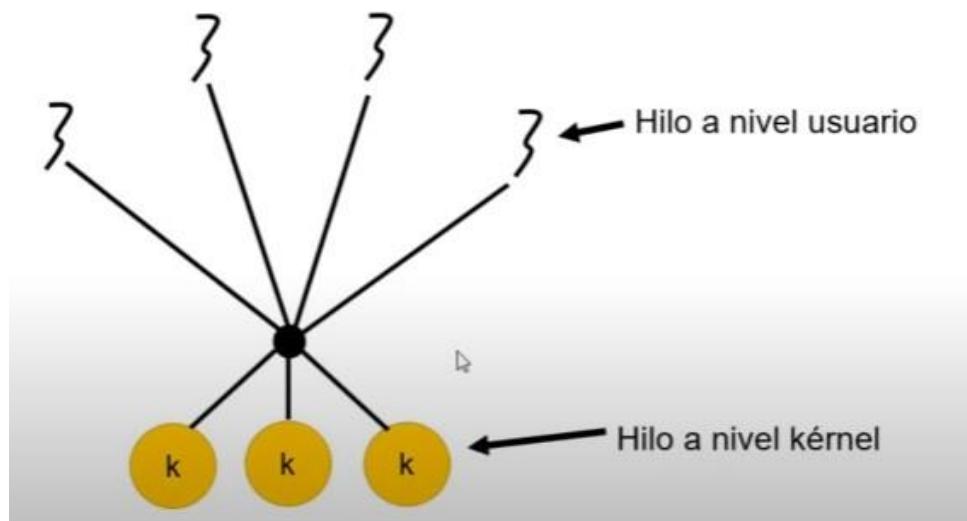
#### Uno a uno

- Cada hilo a nivel usuario corresponde a un hilo del kernel.



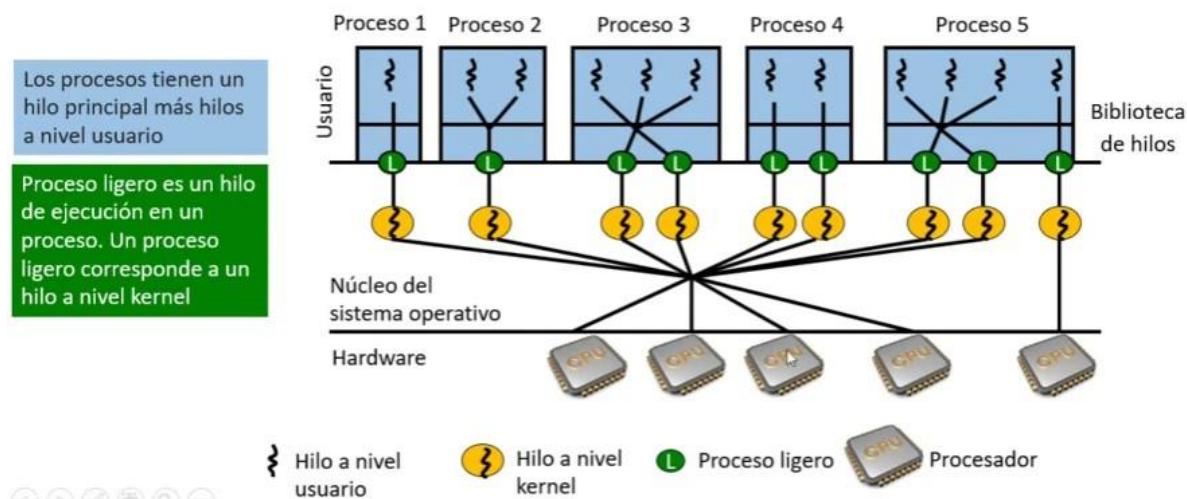
### Muchos a Muchos

#### Modelo muchos a muchos



En este ejemplo de **muchos hilos (nivel usuario)** a **muchos hilos (nivel kernel)** solamente 3 hilos pueden estar ejecutándose porque hay **3 hilos** a nivel kernel disponible.

## Ejemplo de la arquitectura multihilo de Solaris



Mirar a partir de **4:06** del video <https://www.youtube.com/watch?v=zeAkHabyGDo> para ver explicación de este ejemplo.

### YouTube 05: Sistemas Operativos, Hilos 5 Arquitectura Multicore

#### Arquitecturas Multicore

La velocidad del CPU tendía a duplicarse cada 18 meses

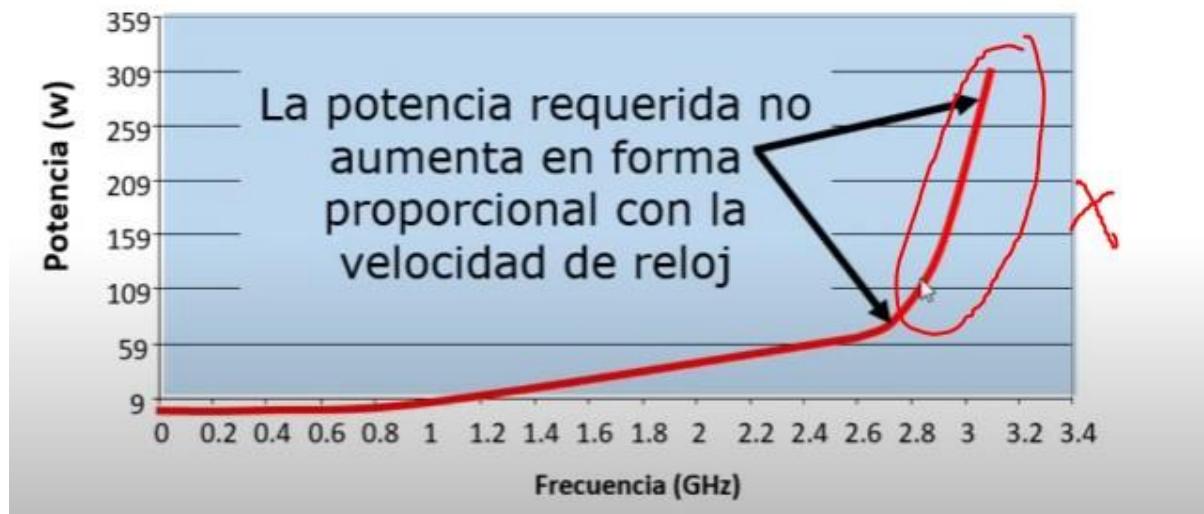
Año	Procesador	Velocidad
1991	Intel 386	25 Mhz
1993	Intel 486	50 Mhz
1995	Pentium	100 Mhz
1996	Pentium MMX	200 Mhz
1998	Pentium II	400 MHz
2000	Pentium III	800 Mhz
2001	Pentium IV	1.6 Ghz
2003	Pentium IV	3.0 Ghz

Desde 2003 hasta hoy en día la velocidad del CPU **NO** se estuvo duplicando cada 18 meses porque llegó a un punto donde la potencia requerida no aumentaba en forma proporcional con la velocidad de reloj del CPU. (**gráfico abajo**).

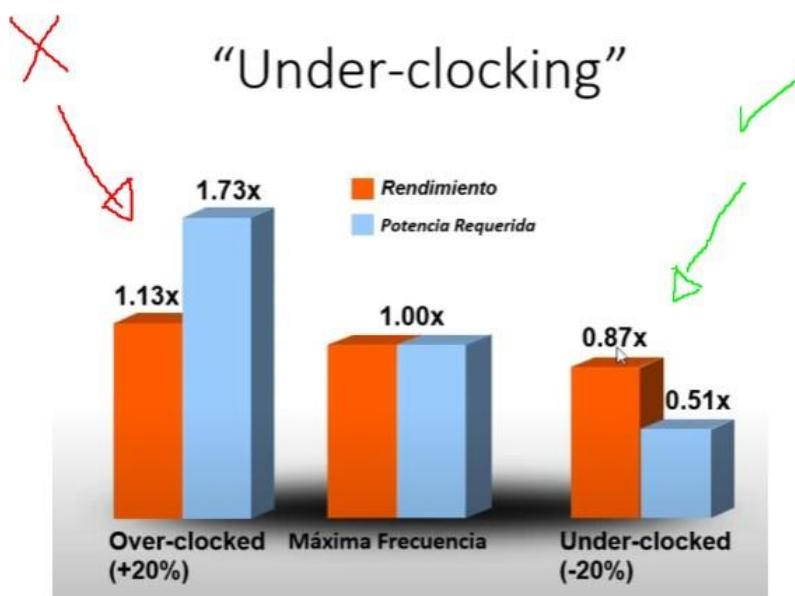
### Potencia y frecuencia

## Potencia y frecuencia

Curva de Potencia vs. Frecuencia para arquitecturas con un núcleo



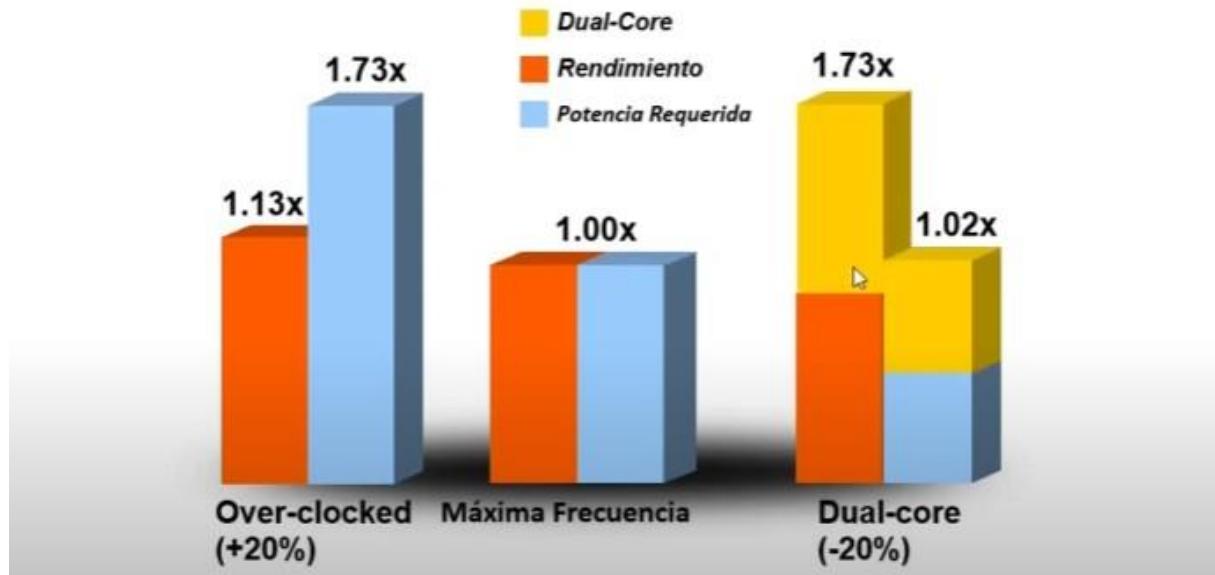
Por lo tanto, **menos velocidad** de reloj nos permite **ahorrar potencia para aumentar los núcleos**.



Aca vemos la potencia requerida para el máximo rendimiento de un núcleo (barra del medio).

En **overclocked** la potencia requerida aumenta un **73%** para una mejora de solamente **13%** en rendimiento, mientras que en **underclocked**, se decrementa **49%** de potencia y disminuye solamente **13%** del rendimiento. A partir de esto, los ingenieros tuvieron esta idea (siguiente gráfico):

## Rendimiento Multi-Core y Consumo de Energía



Ahora con el **underclock** de 2 cores, la potencia requerida aumenta solamente un 2% más que la potencia máxima, y el rendimiento incrementa 73% del rendimiento máximo.

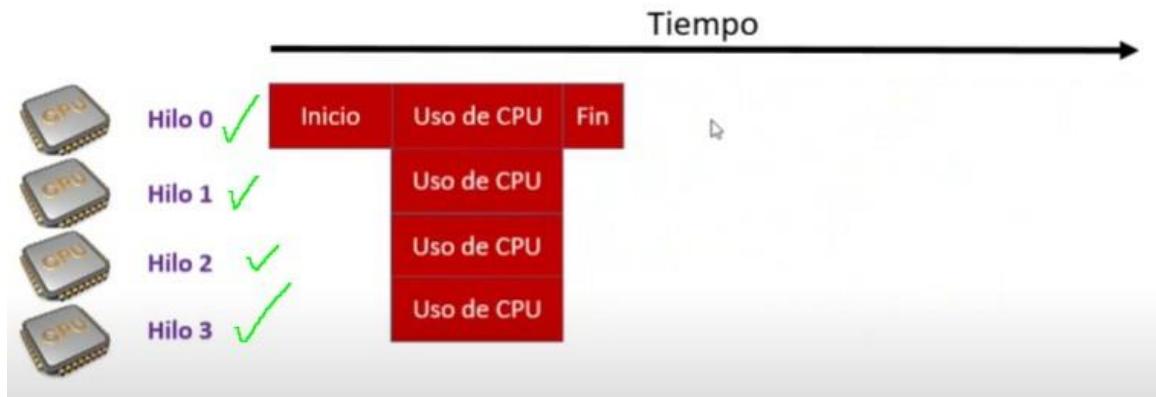
### Aplicación multihilos en una arquitectura con multiple CPU's o Multicore

¿Para que me sirven los hilos en una **arquitectura multicore** o una **arquitectura donde haya múltiples CPU's**?

Con un solo hilo, una aplicación no puede sacar provecho de una arquitectura donde haya múltiples CPU's disponible (multicore)



Pero si dividimos ese mismo proceso de 1 hilo y lo convertimos en un proceso con varios hilos, cada hilo puede ejecutar una parte del proceso en un CPU (core) diferente, se puede aprovechar el paralelismo.



### YouTube 06: Sistemas Operativos, Hilos 6 Hilos Posix

#### Gestión de hilos básica

##### Un hilo tiene:

- Un identificador (TID)
- Un stack
- Una prioridad de ejecución
- Una dirección de inicio de ejecución

#### Gestion de hilos basica (POSIX)

Se dice que un hilo es dinámico si se puede crear en cualquier instante durante la ejecución.

##### En POSIX:

- Los hilos se crean dinámicamente con la función `pthread_create`
- Un hilo termina al invocar la función `pthread_exit`
- El hilo principal debe esperar a que todos los hilos terminen, utilizando la función `pthread_join`

#### Ejemplo hilos Posix

```
void *tfunc(void *args)
{
    printf("Hola mundo\n");
}

int main()
{
    int i;
    pthread_t id_hilo[NTHREADS];

    for(i=0;i<3;i++)
        pthread_create(&id_hilo[i],NULL,tfunc,NULL);
    for(i=0;i<3;i++)
        pthread_join(id_hilo[i],NULL);
}
```

El argumento **tfunc** se indica la función a partir de la cual inicia la ejecución del hilo que voy a crear.

```
...:e(&id_hilo[i],N
```

Ese argumento es la variable donde se **guarda el identificador del hilo a crear**.

## Ejemplo hilos Posix

The diagram illustrates a C program for POSIX threads. A legend indicates that green lines represent 'Hilo creado' (created thread) and red lines represent the 'Hilo principal' (main thread). The code shows a function `tfunc` that prints "Hola mundo". In `main()`, it creates three threads using `pthread_create` and joins them using `pthread_join`. A callout box notes that the main thread must wait for all created threads to finish, or the process will terminate if it does not.

```
void *tfunc(void *args)
{
    printf("Hola mundo\n");
}

int main()
{
    int i;
    pthread_t id_hilo[NTHREADS];

    for(i=0;i<3;i++)
        pthread_create(&id_hilo[i],NULL,tfunc,NULL);

    for(i=0;i<3;i++)
        pthread_join(id_hilo[i],NULL);
}
```

— Hilo creado  
— Hilo principal

El hilo principal debe esperar a que terminen todos los hilos creados  
• Si el hilo principal no espera a que terminen los hilos creados, al terminar termina el proceso con todos los hilos.

Espera a que termine el hilo creado

Mirar a partir de 2:11 para ver la explicación de este ejemplo.

### Compilar pthreads

- Para compilar un programa con **pthread** (Posix Threads):

```
$ gcc -o programa programa.c -lpthread
```

Necesario encadenar con la librería pthread

## Itinerario 6: Mecanismos de comunicación y sincronización

### Condiciones de carrera

El funcionamiento de un proceso y su resultado debe ser independiente de su velocidad relativa de ejecución con respecto a otros procesos, por lo que es necesario garantizar que el orden de ejecución no afecte al resultado. Por medio de la Exclusión Mutua se logra conseguir que un conjunto de instrucciones se ejecute de forma atómica.

Se debe determinar la Sección crítica, que es el segmento de código que manipula un recurso y debe ser ejecutado de forma atómica; para ellos se asocia a un recurso un mecanismo de gestión de exclusión mutua y solamente un proceso puede estar simultáneamente en la sección crítica de un recurso.

Nota: si no se implementa adecuadamente pueden generar problemas de Interbloqueos (Deadlocks)

### Mecanismos de comunicación

Los mecanismos de comunicación permiten la transferencia de información entre dos procesos.

- Archivos
- Tuberías (pipes, FIFO)
- Variables en memoria compartida
- Paso de mensajes

### Mecanismos de sincronización

Los mecanismos de sincronización permiten forzar a un proceso a detener su ejecución hasta que ocurra un evento en otro proceso.

- Construcciones de los lenguajes concurrentes (procesos ligeros)
- Servicios del sistema operativo:
  - Señales (asincronismo)
  - Tuberías (pipes, FIFO)
  - Semáforos
  - Mutex y variables condicionales
  - Paso de mensajes

### Procesos concurrente

- Dos procesos son concurrentes cuando se ejecutan de manera que sus intervalos de ejecución se solapan.



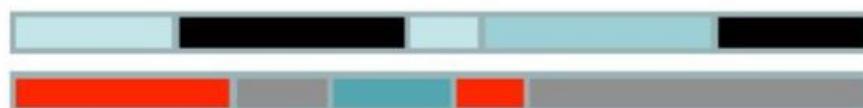
### Tipos de concurrencia

- Concurrencia aparente: Hay más procesos que procesadores.
  - Los procesos se multiplexan en el tiempo.
  - Pseudoparalelismo

1 CPU



2 CPUs



Concurrencia real: Cada proceso se ejecuta en un procesador.

- Se produce una ejecución en paralelo.
- Paralelismo real.

4 CPUs



### Modelos de programación concurrente

- **Multiprogramación** con un único procesador
  - El sistema operativo se encarga de repartir el tiempo entre los procesos (planificación expansiva/no expansiva).
- **Multiprocesador**
  - Se combinan paralelismo real y pseudoparalelismo.
  - Normalmente más procesos que CPU's.
- **Sistema distribuido**
  - Varios computadores conectados por red.

### Ventajas de la ejecución concurrente

- **Facilita la programación.**
  - Diversas tareas se pueden estructurar en procesos separados.
  - Servidor Web: Un proceso encargado de atender a cada petición.
- **Acelera la ejecución de cálculos.**
  - División de cálculos en procesos ejecutados en paralelo.
  - Ejemplos: Simulaciones, Mercado eléctrico, Evaluación de carteras financieras.
- **Mejora la interactividad de las aplicaciones.**
  - Se pueden separar las tareas de procesamiento de las tareas de atención de usuarios.
  - Ejemplo: Impresión y edición.
- **Mejora el aprovechamiento de la CPU.**
  - Se aprovechan las fases de E/S de una aplicación para procesamiento de otras.

### Tipos de procesos concurrentes

#### Independientes.

- **Procesos que se ejecutan concurrentemente pero sin ninguna relación.**
  - No necesitan comunicarse.
  - No necesitan sincronizarse.
  - Ejemplo: Dos intérpretes de mandatos de dos usuarios ejecutados en distintos terminales.

#### Cooperantes.

- **Procesos que se ejecutan concurrentemente con alguna interacción entre ellos.**
  - Pueden comunicarse entre sí.
  - Pueden sincronizarse.
  - Ejemplo: Servidor de transacciones organizado en proceso receptor y procesos de tratamiento de peticiones

**• Acesso a recursos compartidos.**

- Procesos que comparten un recurso.
- Procesos que compiten por un recurso.
- Ejemplo: Servidor de peticiones en la que distintos procesos se escriben en un registro de actividad (log).

**• Comunicación.**

- Procesos que intercambian información.
- Ejemplo: Receptor de peticiones debe pasar información a proceso de tratamiento de petición.

**• Sincronización.**

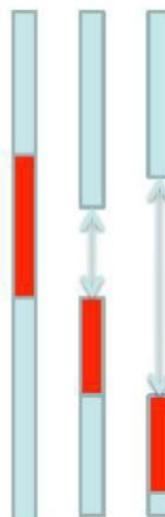
- Un proceso debe esperar a un evento en otro proceso.
- Ejemplo: Un proceso de presentación debe esperar a que todos los procesos de cálculo terminen.

**Exclusión mutua**

- **Sección crítica:** Segmento de código que manipula un recurso y debe ser ejecutado de forma atómica.

- Se asocia a un recurso un mecanismo de gestión de exclusión mutua.

- Solamente un proceso puede estar simultáneamente en la sección crítica de un recurso.



**Problemas de la sección crítica**

**• Interbloqueos.**

- Se produce al admitirse exclusión mutua para más de un recurso.
  - El proceso P1 entra en la sección crítica para el recurso A.
  - El proceso P2 entra en la sección crítica para el recurso B.
  - El proceso P1 solicita entrar en la sección crítica para el recurso B (queda a la espera de que P2 la abandone).
  - El proceso P2 solicita entrar en la sección crítica para el recurso A (queda a la espera de que P1 la abandone).

**Ninguno puede avanzar**

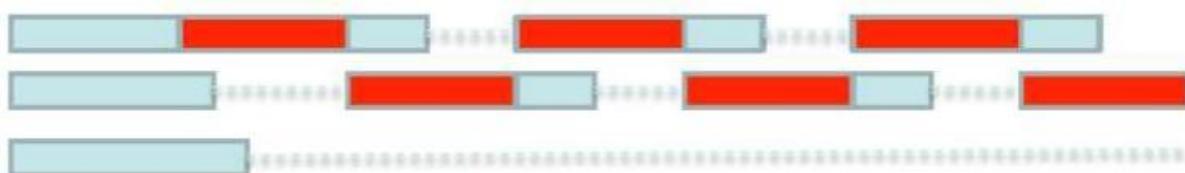
### Problemas de la sección crítica

- Inanición.

– Un proceso queda indefinidamente bloqueado en espera de entrar en una sección crítica.

- El proceso P1 entra en la sección crítica del recurso A.
- El proceso P2 solicita entrar en la sección crítica del recurso A.
- El proceso P3 solicita entrar en la sección crítica del recurso A.
- El proceso P1 abandona la sección crítica del recurso A.
- El proceso P2 entra en la sección crítica del recurso A.
- El proceso P1 solicita entrar en la sección crítica del recurso A.
- El proceso P2 abandona la sección crítica del recurso A.
- El proceso P1 entra en la sección crítica del recurso A.
- ...

**El proceso P3 nunca consigue entrar en la sección crítica del recurso A**



**El proceso P3 nunca llega a conseguir entrar en la sección crítica**

### Condiciones para la exclusión mutua

- Solamente se permite un proceso puede estar simultáneamente en la sección crítica de un recurso.
- No debe ser posible que un proceso que solicite acceso a una sección crítica sea postergado indefinidamente
- Cuando ningún proceso esté en una sección crítica, cualquier proceso que solicite su entrada lo hará sin demora.
- No se puede hacer suposiciones sobre la velocidad relativa de los procesos ni el número de procesadores.
- Un proceso permanece en su sección crítica durante un tiempo finito.

### Sección crítica: Mecanismo de sincronización

- Cualquier mecanismo que solucione el problema de la sección crítica debe proporcionar sincronización entre procesos.
  - Cada proceso debe solicitar permiso para entrar en la sección crítica
  - Cada proceso debe indicar cuando abandona la sección crítica.

Código no crítico

...

**<Entrada en sección crítica>**

Código de sección crítica

**<Salida de sección crítica>**

...

Código no crítico

### Alternativas de implementación

- Desactivar interrupciones.
  - El proceso no sería interrumpido.
  - Solamente sería válido en sistemas monoprocesador.
- Instrucciones máquina.
  - Test and set o swap.
  - Implica espera activa.
  - Son posibles inanición e interbloqueo.
- Otra alternativa: Soporte del sistema operativo.

### Solución de Peterson

SOLO para 2 procesos

- Asume que las instrucciones LOAD y STORE son atómicas, no interrumpibles.
- Los 2 procesos comparten 2 variables:
  - int turno; Boolean flag[2]
- Turno: indica quién entrará en la sección crítica.
- Flag: indica si un proceso está listo para entrar en la sección crítica.
  - flag[i] = true implica que Pi está listo

### Algoritmo para proceso Pi

2 processes: Pi and Pj, where j=1-i

- i = 0 => j=1-i=1
- i = 1 => j=1-i=0

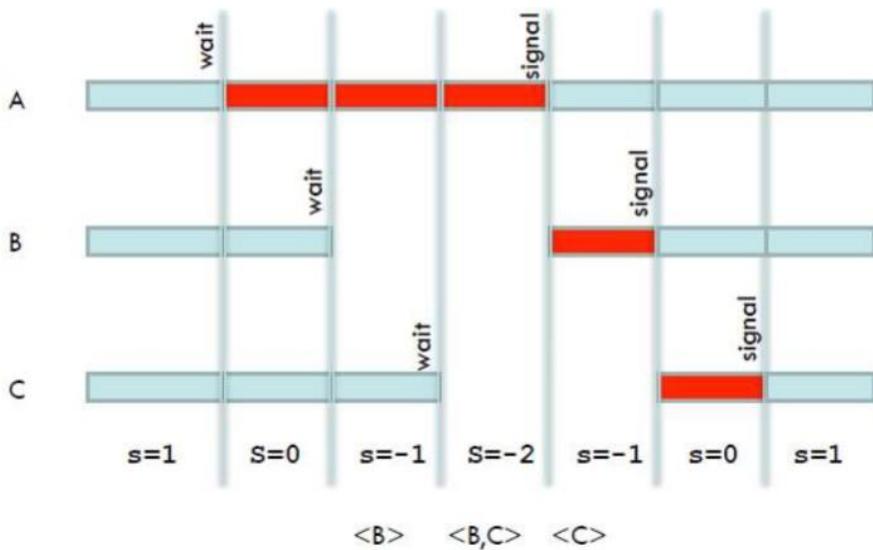
```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[!i] = FALSE;
    remainder section
} while (TRUE);
```

### SEMAFOROS (Dijkstra)

- Sincronización de procesos mediante un mecanismo de señalización a → semáforo.
- Se puede ver un semáforo como una variable entera con tres operaciones asociadas.
  - Iniciación a un valor no negativo.
  - **semWait**: Decrementa el contador del semáforo.
    - Si s<0 → El proceso se bloquea.
  - **semSignal**: Incrementa el valor del semáforo.
    - Si s<=0 → Desbloquea un proceso.

### Operaciones atómicas

### Secciones críticas y semáforos



### El problema del productor - consumidor

- Un proceso produce elementos de información.
- Un proceso consume elementos de información.
- Se tiene un espacio de almacenamiento intermedio.



### Buffer infinito

- Productor
- Consumidor

```

for (;;) {
    x= producir();
    v[fin] = x;
    fin++;
}
    
```

Hay que introducir sincronización

```

for (;;) {
    while (inicio==fin)
        {}
    y=v[inicio];
    inicio++;
    procesar(y);
}
    
```

Espera activa



- Productor

```

semaforo s=1
for (;;) {
    x= producir();
    semWait(s);
    v[fin] = x;
    fin++;
    semSignal(s);
}

```

- Consumidor

```

semaforo s=1
for (;;) {
    while (inicio==fin)
        {}
    semWait(s);
    y=v[inicio]; Espera activa
    inicio++;
    semSignal(s);
    procesar(y);
}

```

- Productor

```

semaforo s=1; semaforo n=0;

for (;;) {
    x= producir();
    semWait(s);
    v[fin] = x;
    fin++;
    semSignal(s);
    semSignal(n)
}

```

- Consumidor

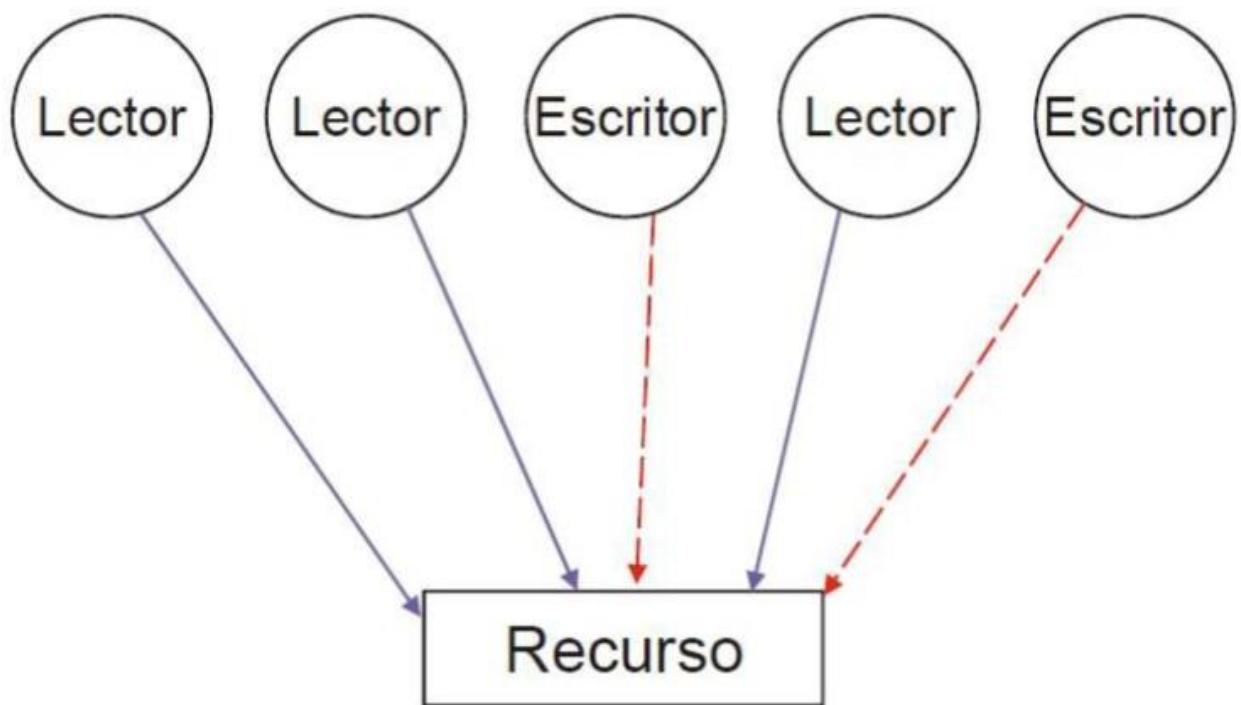
```

int m;
for (;;) {
    semWait(n);
    semWait(s);
    y=v[inicio];
    inicio++;
    semSignal(s);
}

```

### Problema de los lectores-escritores

- Problema que se plantea cuando se tiene un área de almacenamiento compartida.
  - Múltiples procesos leen información.
  - Múltiples procesos escriben información.
- Condiciones:
  - Cualquier número de lectores pueden leer de la zona de datos concurrentemente.
  - Solamente un escritor puede modificar la información a la vez.
  - Durante una escritura ningún lector puede realizar una consulta.



#### Diferencias con otros problemas

- **Exclusión mutua:**

- En el caso de la exclusión mutua solamente se permitiría a un proceso acceder a la información.
- No se permitiría concurrencia entre lectores.

- **Productor consumidor:**

- En el productor/consumidor los dos procesos modifican la zona de datos compartida.

- **Objetivos de restricciones adicionales:**

- Proporcionar una solución más eficiente.

#### Alternativas de gestión

- **Los lectores tienen prioridad.**

- Si hay algún lector en la sección crítica otros lectores pueden entrar.
- Un escritor solamente puede entrar en la sección crítica si no hay ningún proceso.
- Problema: Inanición para escritores.

- **Los escritores tienen prioridad.**

- Cuando un escritor desea acceder a la sección crítica no se admite la entrada de nuevos lectores.

- Lector

```
int nlect; semaforo lec=1; semaforo = escr=1;

for(;;) {
    semWait(lec);
    nlect++;
    if (nlect==1)
        semWait(escr);
    semSignal(lec);

    realizar_lect();

    semWait(lec);
    nlect--;
    if (nlect==0)
        semSignal(escr);
    semSignal(lec);
}
```

**Tarea: Diseñar una solución para escritores con prioridad**

### Antecedentes

Acceso concurrente a datos compartidos puede resultar en inconsistencia de datos.

La consistencia de datos requiere de mecanismos para asegurar la ejecución ordenada de procesos cooperativos.

Buscamos soluciones para el problema del **productor-consumidor**.

Utilizamos contadores.

### Productor

```
while (count == BUFFER_SIZE)
    ; // do nothing

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

### Consumidor

```
while (count == 0)
    ; // do nothing

// remove an item from the buffer
--count;
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

### Condición de concurso

count++ puede implementarse como

```
register1 = count  
register1 = register1 + 1  
count = register1
```

count-- puede implementarse como

```
register2 = count  
register2 = register2 - 1  
count = register2
```

Considera la ejecución alternada, iniciando con "count = 5":

- S0: producer execute register1 = count {register1 = 5}
- S1: producer execute register1 = register1 + 1 {register1 = 6}
- S2: consumer execute register2 = count {register2 = 5}
- S3: consumer execute register2 = register2 - 1 {register2 = 4}
- S4: producer execute count = register1 {count = 6 }
- S5: consumer execute count = register2 {count = 4}

### Solución problema de la sección crítica

**Exclusión Mutua** - Si el proceso Pi está ejecutando en su sección crítica, ningún otro proceso puede entrar a su sección crítica

**Progreso** - Si ningún proceso está ejecutando en su sección crítica y existen procesos que desean entrar a su sección crítica, la selección de los procesos que entrarán a continuación en su sección crítica no puede posponerse indefinidamente

**Espera acotada** - Limitar número de veces que otros procesos son permitidos en sus regiones críticas, antes de que otro proceso pueda entrar.

El problema del **productor / consumidor** está en la operación que realiza el productor cuando le suma 1 al área de memoria compartida al contador, y la operación del consumidor que le resta 1 al área de memoria compartida contador.

La operación de suma se realiza en 3 operaciones de bajo nivel:

**AC ← \*dircontador**

**AC ← AC + 1**

**\*dircontador ← AC**

- La 1º operación es llevar el **contenido del área de memoria contador** al registro **acumulador** del procesador.

**AC ← \*dircontador**

- La 2º operación es **sumarle 1** al registro **acumulador** del procesador, y dejar el resultado en el registro **acumulador** del procesador.

**AC ← AC + 1**

- La 3º operación es llevar el resultado del registro **acumulador** al área de **memoria compartida** contador  
**\*dircontador ← AC**

En el proceso consumidor pasa exactamente lo mismo, la única diferencia es que se le resta 1 al area de memoria compartida contador

**AC ← \*dircontador**

**AC ← AC - 1**

**\*dircontador ← AC**

## Productor Consumidor 01

El productor es un proceso que produce datos en un área compartida y comienza así el código:

### Productor

```
Productor()
{
    pone = 0 ;
    while(true)
    {
        while(contador== BUFFER_SIZE ) ;

        buffer[pone] = Produccion() ;

        contador = contador + 1 ;

        pone = (pone +1) % BUFFER_SIZE ;
    }
}
```

**pone = 0;** **pone** es una variable del productor que le indica en qué posición del buffer tiene que poner el dato, y después entra en un ciclo infinito “**while(true)**” en un ciclo infinito de producción.

**while ( contador == BUFFER\_SIZE );** es un control para que el productor no produzca o deje de producir cuando la cantidad de elementos que produjo, la cantidad de elementos que hay en el buffer, es igual a la cantidad de elementos que el buffer puede contener, entonces si el productor produce, puede destruir algún dato que todavía no haya sido consumido.

**buffer [pone] = Producción();** genera un dato y que se pone en el buffer o en la memoria compartida en la posición “**pone**”.

**contador = contador+1;** el productor registra que acaba de producir un elemento, entonces el contador cuenta la cantidad de elementos que son producidos.

**pone = (pone+1) % BUFFER\_SIZE;** se le asigna un nuevo valor a **pone** para que apunte a la posición siguiente del buffer para que el productor pueda producir casualmente en un área consecutiva de memoria.

### Consumidor

```
Consumidor()
{
    sale = 0 ;
    while(true)
    {
        while(contador== 0 ) ;
        elemento = buffer[sale] ;
        contador= contador - 1 ;
        sale = (sale +1) % BUFFER_SIZE ;
    }
}
```

El proceso consumidor es el proceso que consume los datos producidos por el productor:

**sale = 0;** la variable **sale** en el consumidor le indica al consumidor desde que posición del buffer tiene que consumir, y después entra en un ciclo infinito “**while(true)**” en un ciclo infinito de consumición.

**while ( contador == 0 );** este es el control que tiene el consumidor para que cuando ha consumido toda la producción, deje de consumir. O si no estaría consumiendo elementos ya consumidos o estaría consumiendo basura del buffer.

**elemento = buffer [sale];** “**elemento**” es una variable local del consumidor y toma el elemento que consume desde la posición **sale** de la memoria compartida llamada “**buffer**”.

**contador = contador-1;** el consumidor registra que acaba de consumir un elemento, entonces el contador resta de la cantidad de elementos que hay en el “**buffer**”.

**sale = (sale+1) % BUFFER\_SIZE;** modifica el valor valor de **sale** para que apunte a la posición siguiente del buffer desde donde el consumidor tiene que consumir el elemento.

**Para poder realizar producción / consumición se necesita dos áreas de memoria compartida.**

Un área de **memoria compartida** es el **buffer** - Es donde el productor va a poner la producción, y desde donde el consumidor va a tomar el dato producido.

El otro área de memoria compartida es el **contador** - Va a almacenar la cantidad de elementos que realmente existe en el buffer.

Por ejemplo: Cuando empieza la **producción / consumición**, el **contador es = 0** y el **buffer está vacío**. Luego cuando el productor produzca un elemento, **contador va a ser = 1** y va a haber un elemento en la **posición 0 del buffer**.

Suponiendo que es una producción de letras (el alfabeto) y que se van a poder almacenar como máximo en el buffer **5 letras**.

Quiere decir que cuando **contador = 5**, el productor no puede producir, y cuando el **contador = 0**, el consumidor no puede consumir

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
    int shmidbuffer = shmget(0xa,5,IPC_CREAT|IPC_EXCL|0600);
    int shmidcontador = shmget(0xa,sizeof(int),IPC_CREAT|IPC_EXCL|0600);

    printf("shmidbuffer %d \n", shmidbuffer);
    printf("shmidcontador %d \n", shmidcontador);

    exit(0);
}
```

```
1 #include <stdio.h>
2 #include <sys/ipc.h>
3 #include <sys/shm.h>
4 #include <unistd.h>
5 #include <stdlib.h>
6
7 int main(void)
8 {
9     int shmidbuffer = shmget(0xa, 5, IPC_CREAT|IPC_EXCL|0600);
10    int shmidcontador = shmget(0xa, sizeof(int), IPC_CREAT|IPC_EXCL|0600);
11
12    printf("shmidbuffer %d \n", shmidbuffer);
13    printf("shmidcontador %d \n", shmidcontador);
14
15    exit(0);
16 }
```

El **5** indica la cantidad de bytes que vamos a demandar para el buffer (**productor escribe 5 caracteres**).

El **sizeof(int)** está pidiendo memoria compartida para ‘contador’ para un entero y el sistema nos va a dar 4 bytes.

Las Líneas 13 y 14 simplemente **muestran el resultado de ejecución** de las llamadas al sistema de las líneas 10 y 11.

## C Linux Semaforos System V 01

### Uso de semáforos

Usaremos las llamadas **semget**, **semctl** y **semop**.

**semget** - creamos un conjunto de semáforos.

**semctl** - podemos inicializar y recuperar el valor del semáforo o borrar el semáforo.

**semop** - podemos hacer un **wait** o un **signal** sobre el semáforo.

### NOMBRE

**semget** - obtiene el identificador de un conjunto de semáforos

### SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

### DESCRIPCIÓN

Esta función devuelve el identificador del conjunto de semáforos asociado con el

argumento key. Un nuevo conjunto de nsems semáforos se crea si key tiene el valor IPC\_PRIVATE, o si no hay un conjunto de semáforos asociado a key y el bit IPC\_CREAT vale 1 en semflg (p.ej. semflg & IPC\_CREAT es distinto de cero).

la presencia en semflg de los campos IPC\_CREAT e IPC\_EXCL tiene el mismo papel, con respecto a la existencia del conjunto de semáforos, que la presencia de O\_CREAT y O\_EXCL en el

argumento mode de la llamada del sistema open(2): p.ej., la función semget falla si semflg tiene a 1 tanto IPC\_CREAT como IPC\_EXCL y ya existe un conjunto de semáforos para key.

#### NOMBRE

**semctl** - operaciones de control de semáforos

#### SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semctl(int semid, int semnum, int cmd, ...);
```

#### DESCRIPCIÓN

La función semctl realiza la operación de control especificada por cmd sobre el conjunto de semáforos identificados por semid, o en el semáforo semnum de dicho conjunto. (Los semáforos son numerados comenzando por el 0.)

#### NOMBRE

**semop** - operaciones con semáforos

#### SINTAXIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

#### DESCRIPCIÓN

La función semop realiza operaciones sobre los miembros seleccionados del conjunto de semáforos indicado por semid. Cada uno de los nsops elementos en el array apuntado por sops especifica una operación a ser realizada en un semáforo mediante una estructura sembuf que incluye los siguientes miembros:

```
unsigned short sem_num; /* numero de semaforo */
short sem_op; /* operación sobre el semáforo */
```

short sem\_fig; /\* banderas o indicadores para la operación \*/

Comando **ipcs** en linux

Comando **ipcs -s** en linux muestra solo semáforos

```
salo@DESKTOP-0M99BC2:~$ ipcs
----- Message Queues -----
key      msqid      owner      perms      used-bytes     messages
----- Shared Memory Segments -----
key      shmid      owner      perms      bytes      nattch      status
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
salo@DESKTOP-0M99BC2:~$ export DISPLAY=0:0
salo@DESKTOP-0M99BC2:~$ ipcs -s
----- Semaphore Arrays -----
key      semid      owner      perms      nsems
salo@DESKTOP-0M99BC2:~$
```

**key** - una clave que le da el programador al conjunto de semáforos creados

**semid** - el identificador del conjunto de semáforos que se lo da el sistema operativo

**owner** - nombre del usuario que creó el recurso

**perms** - los permisos de acceso que tienen esos semáforos

**nsems** - muestra cantidad de semáforos que se crearon para el conjunto

El **semget** nos devuelve un entero (int semID) que es el identificador del conjunto de recursos semáforos.

El semget **recibe como argumentos de entrada una clave** (key(0xa)) que es un número hexadecimal otorgada por el programador

El segundo argumento que recibe es la **cantidad de semáforos del conjunto** (por ej: 2) que van a pertenecer al conjunto semID

El tercer argumento son **los flag** ( IPC\_CREAT , IPC\_EXCL , perms )

En este ejemplo de 2 semáforos, le vamos a pasar el permiso **0600** donde el **6** representa el permiso de **Lectura** y permiso de **Escritura**.

```
#include <stdio.h> //cabecera para printf
#include <stdlib.h> //cabecera para exit
#include <errno.h> //cabecera para capturar eventuales errores del semget
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/ipc.h>
```

```
int main(void)
{
    int semid = semget(0xa,2,IPC_CREAT|IPC_EXCL|0600);
    if(errno == EEXIST)
        printf("ya existe un conjunto de semáforos para la clave 0xa\n");
    else
        printf("semid = %d\n",semid);
    exit(0);
}
```

El **semctl** nos devuelve un entero (-1 si hay error).

El semget **recibe como argumentos** un entero (semID).

El segundo argumento que recibe es el **número de semáforos** (en nuestro caso hay 2 semáforos así que le deberíamos pasarle el número de semáforo 0 o 1).

El tercer argumento es un entero (int) comando (cmd) llamado SETVAL, es el comando que voy a usar para setear el valor del semáforo.

En este ejemplo de 2 semáforos, le vamos a pasar el permiso **0600** donde el **6** representa el permiso de **Lectura** y permiso de **Escritura**.

El cuarto argumento es un entero (int) que es el valor inicial del semáforo

```
1 #include <sys/ipc.h>
2 #include <sys/sem.h>
3 #include <sys/ipc.h>
4
5 void P(int semid, int sem)
6 {
7     struct sembuf buf;
8     buf.sem_num = sem;
9     buf.sem_op = -1;
10    buf.sem_flg = 0;
11    semop(semid,&buf,1);
12 }
13
14 void V(int semid, int sem)
15 {
16     struct sembuf buf;
17     buf.sem_num = sem;
18     buf.sem_op = 1;
19     buf.sem_flg = 0;
20     semop(semid,&buf,1);
21 }
```

Cómo compilar un código que no es main

```
salo@DESKTOP-0M99BC2:~$ cc PyV.c -Wall -g -c  
salo@DESKTOP-0M99BC2:~$
```

Ver minuto **36:00** del video para ver explicación de los dos semáforos creados.

#### Comandos para compilación

`cc PyV.c -Wall -g -c` //línea de comando para obtener el programa objeto P y V .c

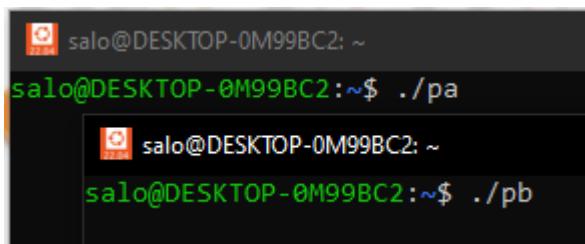
`cc pa.c -Wall -g -c` //para obtener el programa objeto de pa.c

`cc pb.c -Wall -g -c` //para obtener el programa objeto de pb.c

`gcc -Wall -g PyV.o pa.o -o pa` //para obtener una ejecutable del objeto PyV.o y objeto pa.o

`gcc -Wall -g PyV.o pb.o -o pb` //para obtener una ejecutable del objeto PyV.o y objeto pb.o

Ejecutamos los procesos pa y pb en diferentes terminales para ver cómo están sincronizado



```
salo@DESKTOP-0M99BC2: ~  
salo@DESKTOP-0M99BC2:~$ ./pa  
salo@DESKTOP-0M99BC2: ~  
salo@DESKTOP-0M99BC2:~$ ./pb
```

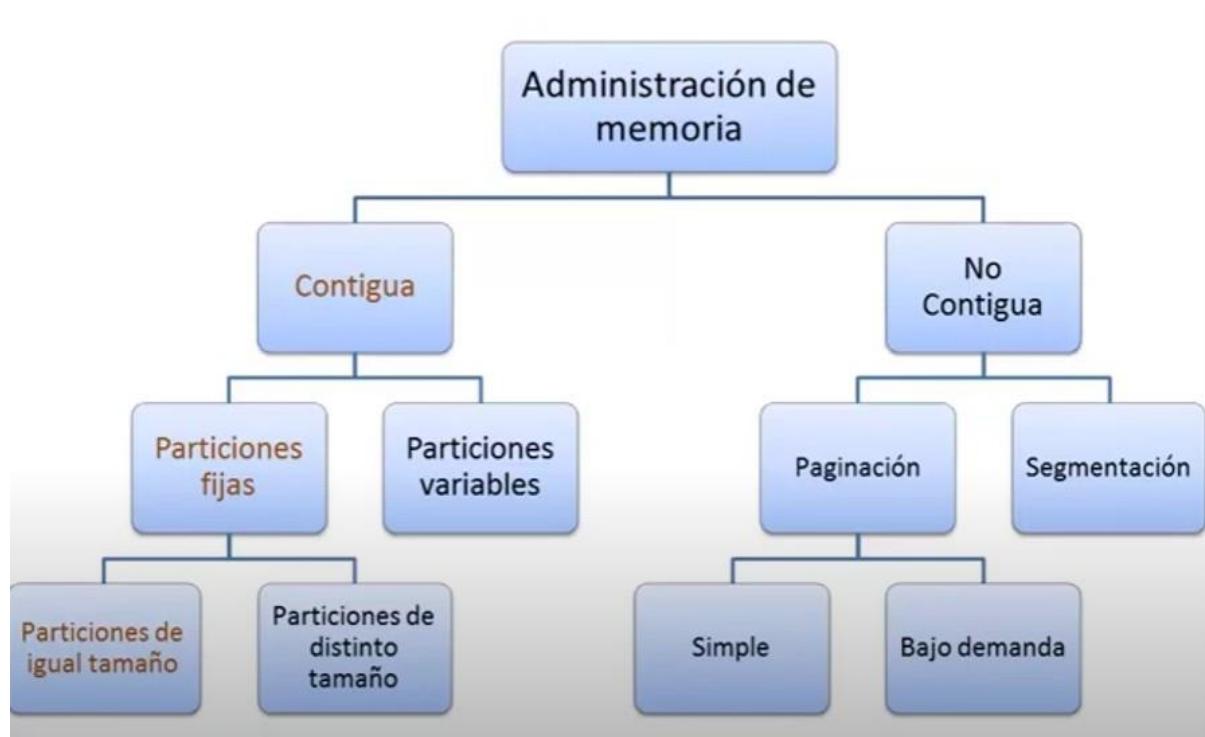
## Unidad 3: Mecanismos de Gestión de Memoria

### Itinerario 10: Manejo de Memoria

#### Objetivos del gestor de memoria

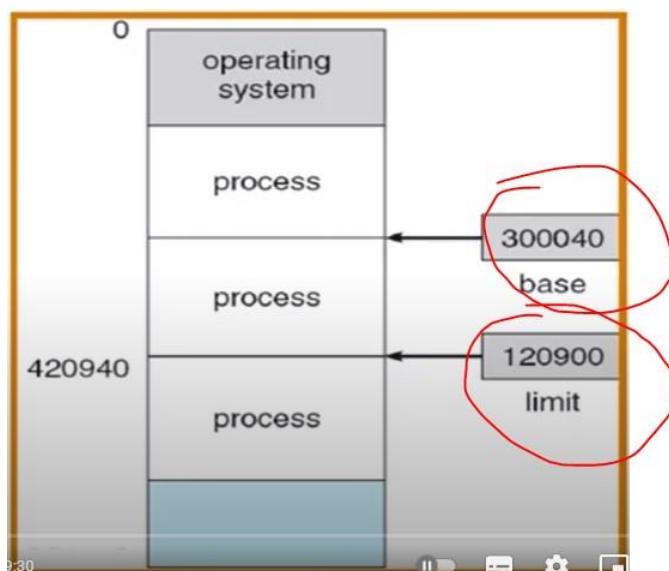
- Ofrecer a cada proceso un espacio lógico propio.
- Proporcionar protección entre procesos.
- Permitir que los procesos compartan memoria.
- Dar soporte a las regiones del proceso.
- Maximizar el grado de multiprogramación.
- Proporcionar a los procesos mapas de memoria muy grande.

## SO Simulación de memoria contigua con particiones de igual tamaño 01



### Registros base y límite

- Un par de registros **base** y **límite** definen el espacio lógico de direcciones



Registro **base** - apunta la dirección de comienzo de la partición  
Registro **límite** - tiene el tamaño del proceso.

**base + límite** = espacio de direcciones lógicas al cual se pueden ir generando direccionamiento durante la ejecución del proceso

(**300,040 + 120,900 = 420,940** → **desde dirección lógico nro. 300,040 hasta dirección lógico nro. 420,940**)

La administración de memoria contigua con particiones fijas de igual tamaño divide la memoria de usuario en porciones de **memorias fijas** y de **igual tamaño**.

**FIJOS** quiere decir que no se puede cambiar el tamaño de las particiones en tiempo de ejecución. (van a ser fijos durante toda la sesión del sistema).

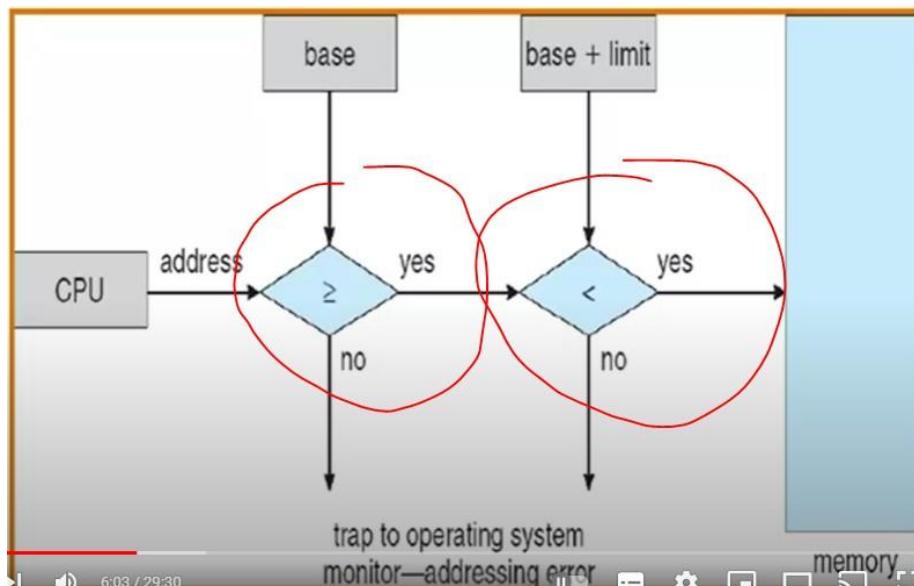
En cada partición solamente puede agregar o almacenar un proceso de usuario y por supuesto el **tamaño del proceso del usuario no puede superar el espacio de direcciones de la partición**. O si no, el proceso es rechazado.

En cambio, si el tamaño del proceso o el **límite del proceso es menor** al tamaño de la partición, en esa partición, va a quedar un hueco de memoria libre que no va a poder ser asignada a ningún proceso.

A ese espacio libre (**hueco**) que queda dentro de la partición que el proceso ocupa se denomina **fragmentación interna**.

**Fragmentación externa** - Todas las particiones que no fueron usadas.

## Protección de direcciones de HW con registros **base** y **límite**



Si la dirección generada por el proceso es menor a la **base**, quiere decir que **no** está dentro de un espacio de direcciones válido.

Si la dirección generada por el proceso es mayor a la **base + límite**, quiere decir que también **no** está dentro de un espacio de direcciones válido.

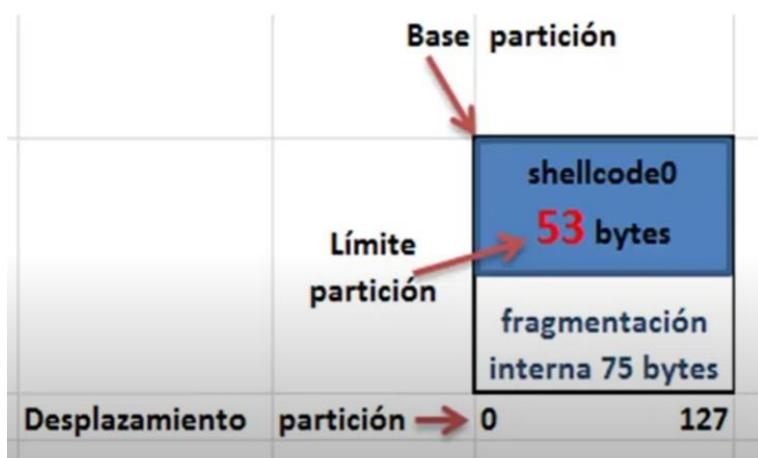
## Simulación de memoria contigua con particiones fijas de igual tamaño

- Memoria principal 512 bytes
- Cuatro particiones de 128 bytes cada una
- Tres particiones ocupadas y una partición libre

Particion 0	Particion 1	Particion 2	Partition 3
ocupada	ocupada	ocupada	libre
0 127	0 127	0 127	0 127
0			511

Particion 0	Particion 1	Particion 2	Partition 3
Shellcode0 53 bytes	Shellcode1 63 bytes	Shellcode2 73 bytes	libre
Fragmentacion interna 75 bytes	Fragmentacion interna 65 bytes	Fragmentacion interna 55 bytes	
0 127	0 127	0 127	0 127
0			511

### Datos importantes



## Información que se debe tener en cuenta en la administración de memoria contigua con particiones fijas de igual tamaño

- Tamaño de la partición
- Dirección de memoria de cada partición (base)
- Límite de cada partición.
- Estado de cada partición (ocupada o libre)
- Fragmentación de cada partición
- Fragmentación externa de la memoria

## Simulacion

- Memoria compartida de 512 bytes
- Dividida lógicamente en cuatro partes (cuatro punteros)
- En la parte 0 guardamos shellcode0
- En la parte 1 guardamos shellcode1
- En la parte 2 guardamos shellcode2
- Un proceso padre crea tres hijos
- Cada hijo pone en ejecución un shellcode
- El hijo 1 ejecuta el shellcode0 de la partición 0
- El hijo 2 ejecuta el shellcode1 de la partición 1
- El hijo 3 ejecuta el shellcode2 de la partición 2

**Paso 1:** crear los tres shellcode en Assembler...

La tarea de shellcode 0 va a ser mostrar \*\*\*\*\* en pantalla  
La tarea de shellcode 1 va a ser mostrar ----- en pantalla  
La tarea de shellcode 2 va a ser mostrar ..... en pantalla

**Paso 2:** obtener el shellcode

### Script para obtener el shellcode0

```
1 nasm -f elf proc00.asm
2 ld -N proc00.o -o proc00
3 objcopy -j .text -O binary proc00.o proc00.bin
4 hexdump -v -e "'\"\\x02x"' proc00.bin > proc00.cod
5 objdump -d -M intel proc00 > proc00.obj
```

```
jromer@debianPGhost1:~/shellcode/shellcode_1$ ls -l *.cod
-rw-r--r-- 1 jromer jromer 220 sep 27 15:53 proc00.cod ←
-rw-r--r-- 1 jromer jromer 256 sep 26 11:52 proc01.cod ←
-rw-r--r-- 1 jromer jromer 296 sep 26 11:52 proc02.cod ←
jromer@debianPGhost1:~/shellcode/shellcode_1$
```

Paso 3: ejecutamos...

```
jromer@debianPGhost1:~/shellcode/shellcode_1$ ./simulaadmcontigua
shmid = 1966093
dirshm = 0xb788f000
Proceso [3101]
-----
Proceso [3100] ←
-----
Proceso [3099] ←
*****
Datos que muestran como fue usada la memoria compartida
dirComienzoParticion[0] = 0xb788f000
dirComienzoParticion[1] = 0xb788f080
dirComienzoParticion[2] = 0xb788f100
dirComienzoParticion[3] = 0xb788f180
limite ShellCode[0] = 53 ←
limite ShellCode[1] = 63 ←
limite ShellCode[2] = 73 ←
fragInternaParticion[0] = 75 ←
fragInternaParticion[1] = 65 ←
fragInternaParticion[2] = 55 ←
fragExterna = 128
```

### Espacios Lógicos Independientes

- No se conoce la posición de memoria donde un programa ejecutará.
- Código en ejecutable genera referencias entre 0 y N.
- Ejemplo: Programa que copia un vector.

En la memoria física de un computador coexisten el sistema operativo, las rutinas de enlace dinámico y los programas de usuario. En los sistemas operativos modernos la gestión de memoria **resuelve aspectos** como:

- La carga de programas y su ubicación. Hay que establecer la correspondencia entre las direcciones lógicas del programa y su ubicación física en memoria.
- La presencia simultánea de más de un programa en memoria.
- La posibilidad de cargar rutinas en tiempo de ejecución (rutinas de enlace dinámico).
- La compartición de espacios de memoria por varios programas.
- La ejecución de programas que no caben completos en memoria.
- La gestión eficiente del espacio de memoria libre.

**Memoria principal.** – Memoria volátil (datos no persistentes), los datos son accedidos por el procesador.

**Memoria secundaria.** – Memoria no volátil (datos persistentes). Es organizada en bloques de datos, pero para simplificar se crean estructuras de archivos y directorios.

*En un sistema operativo multiprogramado, no se pueden colocar todos los programas a partir de la misma dirección (p. ej. 0). En consecuencia existe la necesidad de poder reubicar un programa a partir de una dirección de memoria.*

**Reubicación:** Traducción de direcciones lógicas en direcciones físicas.

Crea espacio lógico independiente para el proceso: El sistema operativo debe poder acceder a espacios lógicos de los procesos.

Ejemplo: Programa tiene asignada memoria a partir de 10000

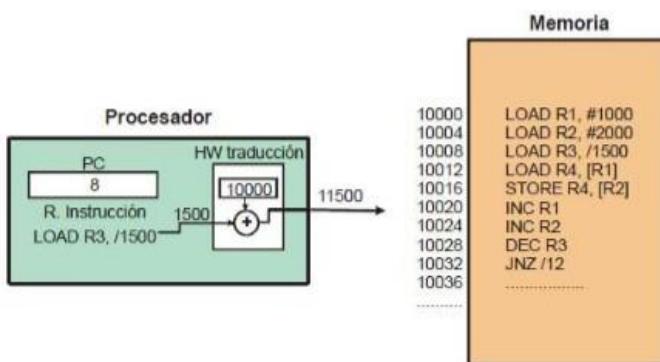
- Sumar 10000 a direcciones generadas

Dos alternativas:

- **Reubicación hardware**
- **Reubicación software**

### Reubicación Hardware

- Hardware (MMU) encargado de traducción
- S.O. se encarga de:
  - Almacena por cada proceso su función de traducción
  - Específica al hardware que funcione aplicar para cada proceso
- Programa se carga en memoria sin modificar



### Reubicación Software

- Traducción de direcciones durante carga del programa
- Programa en memoria es distinto del ejecutable
- **Desventajas:**
  - No asegura protección
  - No permite mover el programa en tiempo de ejecución

## Protección

- **Monoprogramación:** Protección del SO
- **Multiprogramación:** Además procesos entre sí
- Traducción debe crear espacios disjuntos
- Necesario validar todas las direcciones que genera el programa
  - La detección debe realizarla el hardware del procesador
  - El tratamiento lo hace el SO
- En sistemas con mapa de E/S y memoria común:
  - Traducción permite impedir que procesos accedan directamente a dispositivos de E/S

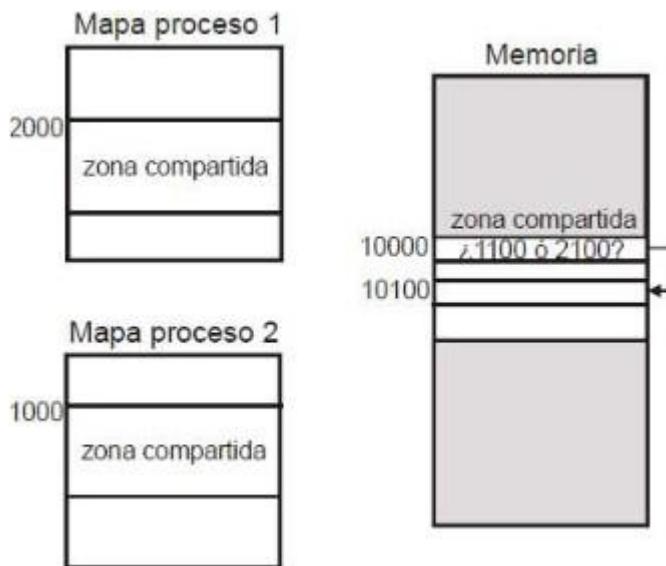
## Comparación de Memoria

- Las direcciones lógicas de 2 o más procesos se corresponden con la misma dirección física.
- Bajo control del S.O.
- **Beneficios**
  - Los procesos ejecutando el mismo programa comparten su código.
  - Mecanismos de comunicación entre procesos muy rápido
- Requiere asignación **no contigua**



## Problemas de compartición de memoria

- Si posición de zona compartida contiene referencia a otra posición de la zona.
- Ejemplo con zonas de código:
  - Zona compartida contiene instrucción de bifurcación a instrucción dentro de la zona.
- Ejemplo con zonas de datos:
  - Zona contiene una lista con punteros



## Soporte de regiones

- Mapa de proceso no homogéneo
  - Conjunto de regiones con distintas características
  - Ejemplo: Región de código no modificable
- Mapa de proceso dinámico
  - Regiones cambian de tamaño (p.ej. pila)
  - Se crean y destruyen regiones
  - Existen zonas sin asignar (huecos)
- Gestor de memoria debe dar soporte a estas características:
  - Detectar accesos no permitidos a una región
  - Detectar accesos a huecos
  - Evitar reservar espacio para huecos
- S.O. debe guardar una tabla de regiones para cada proceso

## Maximización del Rendimiento

- Reparto de memoria maximizando grado de multiprogramación
- Se “desperdicia” memoria debido a:
  - “Restos” inutilizables (fragmentación)
  - Tablas requeridas por gestor de memoria
- Menor fragmentación → Tablas más grandes
- Compromiso: Paginación
- Uso de memoria virtual para aumentar grado de multiprogramación

### Páginas de una dirección

- No hay fragmentación
- Irrealizable por tamaño de TP

Memoria

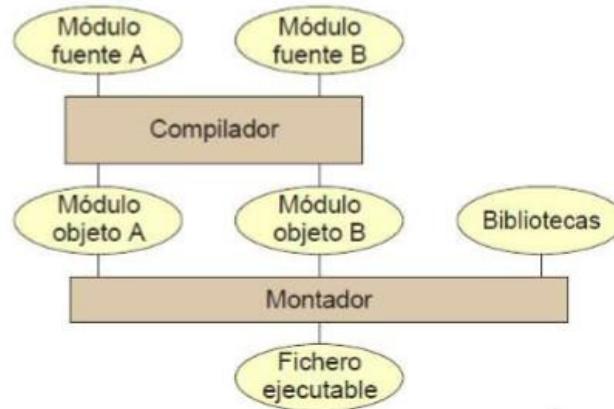
	Memoria
0	Dirección 50 del proceso 4
1	Dirección 10 del proceso 6
2	Dirección 95 del proceso 7
3	Dirección 56 del proceso 8
4	Dirección 0 del proceso 12
5	Dirección 5 del proceso 20
6	Dirección 0 del proceso 1
.....	
.....	
N-1	Dirección 88 del proceso 9
N	Dirección 51 del proceso 4

## Mapas de memorias muy grandes

- Procesos necesitan cada vez mapas más grandes
  - Aplicaciones más avanzadas o novedosas
- Resuelto gracias al uso de memoria virtual
- Históricamente se han usado overlays:
  - Programa dividido en fases que se ejecutan sucesivamente
  - En cada momento sólo hay una fase residente en memoria
  - Cada fase realiza su labor y carga la siguiente
  - No es transparente: Toda la labor realizada por programador

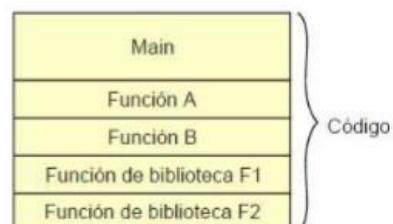
## Fases de generación de un ejecutable

- Aplicación: conjunto de módulos en lenguaje de alto nivel
- Procesado en dos fases: Compilación y Montaje
- Compilación:
  - Resuelve referencias dentro cada módulo fuente
  - Genera módulo objeto
- Montaje (o enlace):
  - Resuelve referencias entre módulos objeto
  - Resuelve referencias a símbolos de bibliotecas
  - Genera ejecutable incluyendo bibliotecas



## Bibliotecas de objetos (y desventajas)

- Biblioteca: colección de módulos objeto relacionados
- Bibliotecas del sistema o creadas por el usuario
- Bibliotecas Estáticas:
  - Montaje: enlaza los módulos objeto del programa y de las bibliotecas
  - Ejecutable autocontenido
- Desventajas del montaje estático:
  - Ejecutables grandes
  - Código de función de biblioteca repetido en muchos ejecutables
  - Múltiples copias en memoria del código de función de biblioteca



## Bibliotecas Dinámicas: Ventajas e Inconvenientes

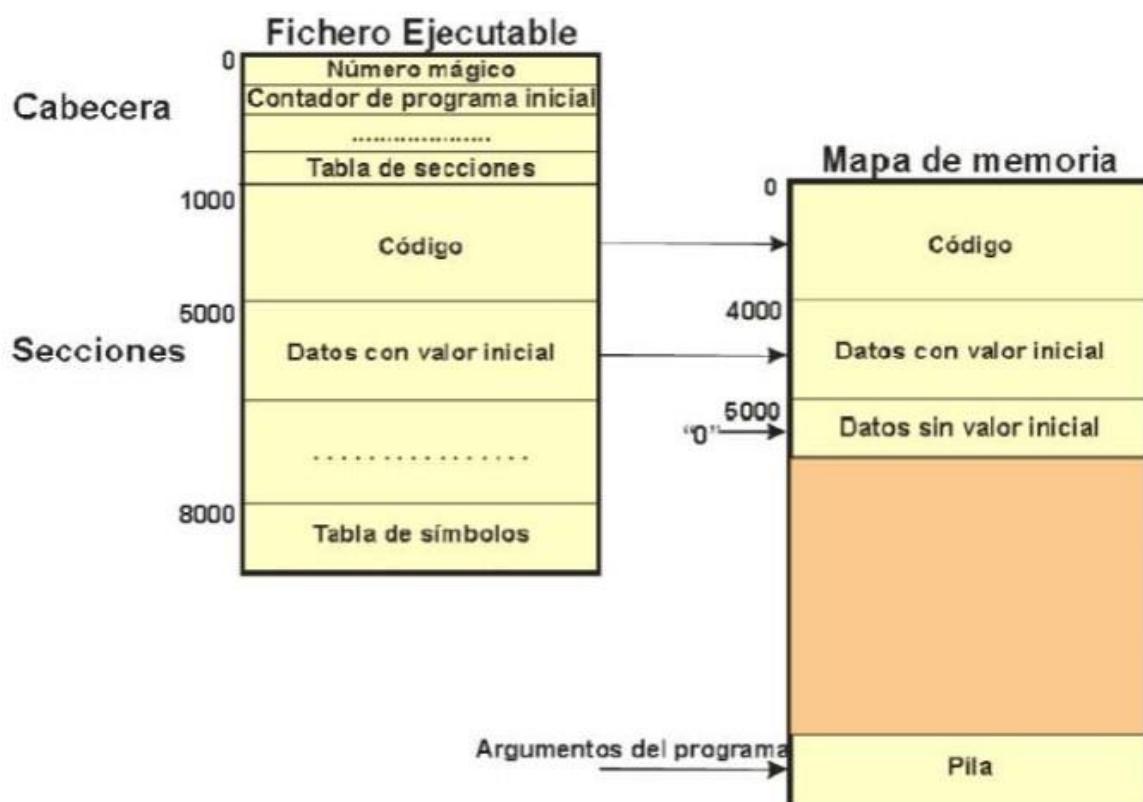
- Ventajas:
  - Menor tamaño ejecutables
  - Código de rutinas de biblioteca sólo en archivo de biblioteca
  - Procesos pueden compartir código de biblioteca
  - Actualización automática de bibliotecas: Uso de versiones
- Inconvenientes:
  - Mayor tiempo de ejecución debido a carga y montaje
    - Tolerable: Compensa el resto de las ventajas
  - Ejecutable no autocontenido
- Uso de bibliotecas dinámicas es transparente
  - Mandatos de compilación y montaje igual que con estáticas

## Mapa de Memoria de un proceso

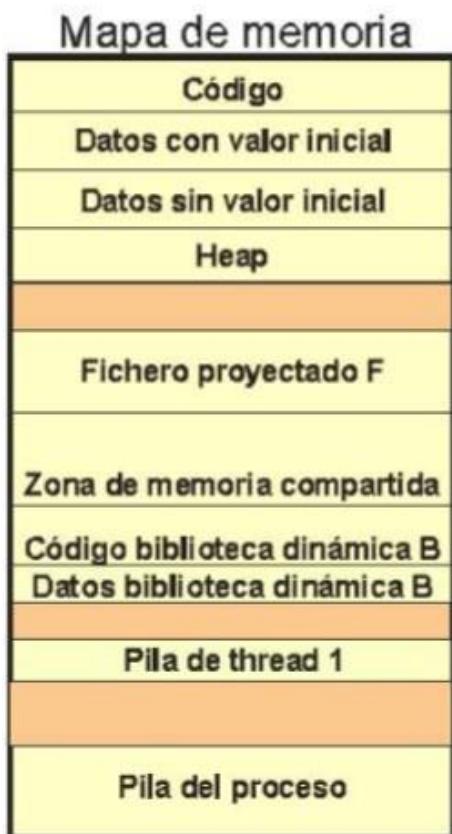
- Mapa de memoria o imagen del proceso: conjunto de regiones
- Región:
  - Tiene asociada una información (un “objeto de memoria”)
  - Zona contigua tratada como unidad al proteger o compartir
- Cada región se caracteriza por:
  - dirección de comienzo y tamaño inicial
  - soporte: donde se almacena su contenido inicial
  - protección: RWX
  - uso compartido o privado
  - tamaño fijo o variable

## Creación de MAPA de memoria inicial a partir de ejecutable

- Ejecución de un programa: Crea mapa a partir de ejecutable
  - Regiones de mapa inicial → Secciones de ejecutable
- Código
  - Compartida, RX, T. Fijo, Soporte en Ejecutable
- Datos con valor inicial
  - Privada, RW, T. Fijo, Soporte en Ejecutable
- Datos sin valor inicial
  - Privada, RW, T. Fijo, Sin Soporte (rellenar 0)
- Pila
  - Privada, RW, T. Variable, Sin Soporte (rellenar 0)
  - Crece hacia direcciones más bajas
  - Pila inicial: argumentos del programa



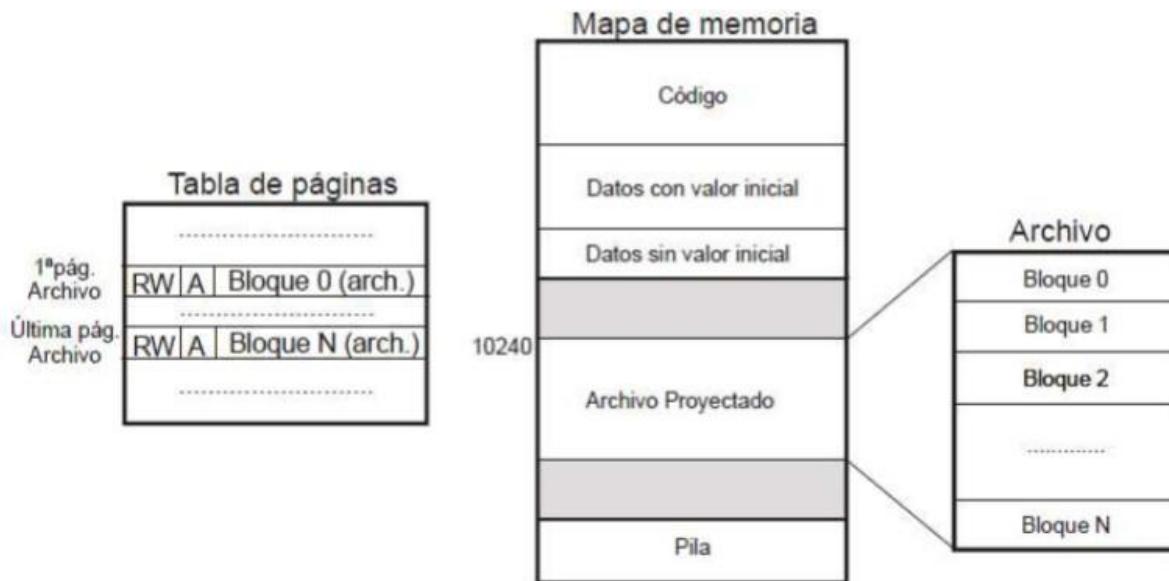
## Possible MAPA de memoria de un proceso



Para estudiar la evolución del MAPA de memoria se pueden distinguir las siguientes operaciones:

- Crear región
  - Implícitamente al crear mapa inicial o por solicitud del programa en t. de ejecución (p.ej. proyectar un archivo)
- Eliminar región
  - Implícitamente al terminar el proceso o por solicitud del programa en t. de ejecución (p.ej. desproyectar un archivo)
- Cambiar tamaño de la región
  - Implícitamente para la pila o por solicitud del programa para el heap
- Duplicar región
  - Operación requerida por el servicio FORK de POSIX

## Proyección de memoria



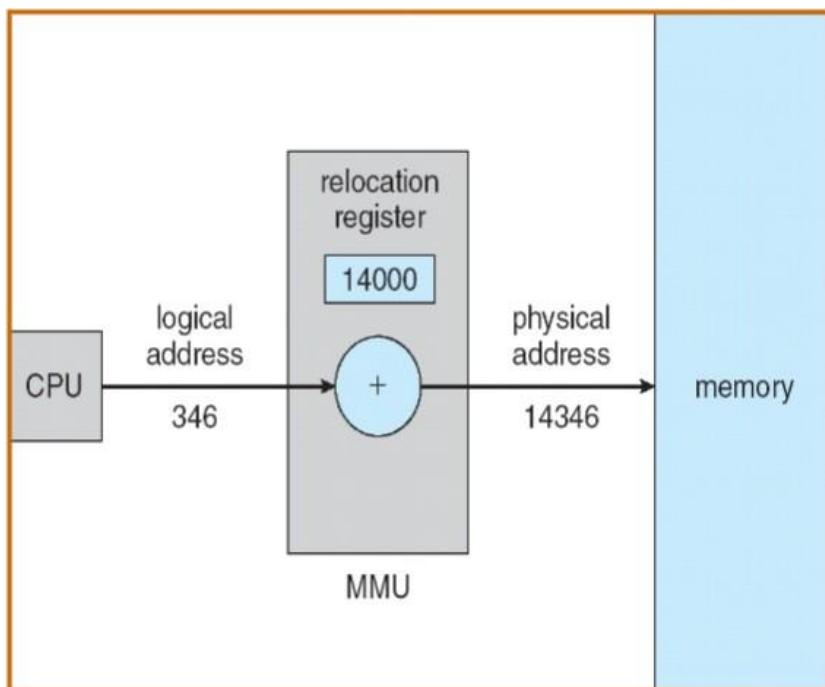
**Dir. lógicas:** direcciones de memoria generadas por el programa

**Dir. físicas:** direcciones de memoria principal asignadas al proceso

Función de traducción: Traducción (IdProc, dir\_logica) --> dir\_fisica

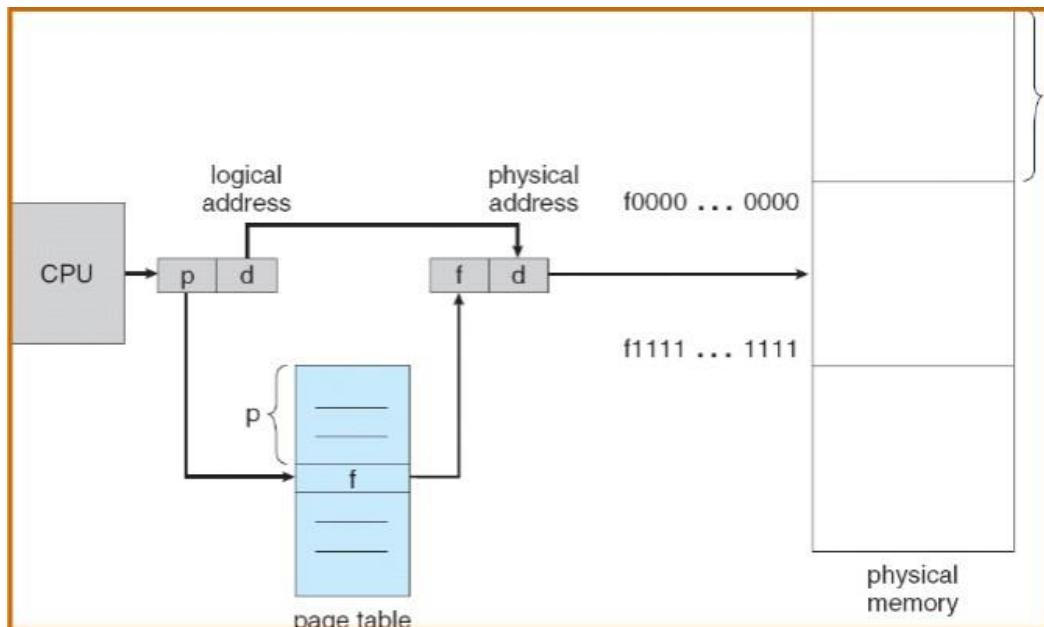
## Unidad de Manejo de Memoria (MMU)

- Dispositivo de hardware que mapea direcciones virtuales a físicas
- En el esquema MMU, el valor en el registro de relocalización **se añade** a cada dirección generada por un proceso de usuario al momento de enviarla a la memoria
- El programa de usuario trabaja con direcciones **lógicas**; nunca ve las direcciones físicas **reales**



### Esquema de traducción de direcciones

- Las direcciones generadas por CPU divididas en:
    - **Página número ( $p$ )** – utilizado como un índice en una *tabla de páginas* que contiene las direcciones base de cada página en memoria física
    - **Desplazamiento de página ( $d$ )** – combinado con direcciones base para definir la dirección de memoria física que es enviada a la unidad de memoria
- | página<br>número | desplazamiento<br>página |
|------------------|--------------------------|
| $p$              | $d$                      |
| $m - n$          | $n$                      |
- Para un espacio de direcciones lógicas  $2^m$  y tamaño de página  $2^n$



Es necesario **soporte hardware** para la traducción de direcciones. Las direcciones lógicas generadas por el programa se **dividen en número de página** (o segmento) y **desplazamiento dentro de la página**. La traducción del número de página (o segmento) a marco de página en memoria física (o dirección de comienzo del segmento) **se realiza en tiempo de ejecución mediante una tabla de páginas** (o tabla de segmentos) **asociada al programa, que habitualmente reside en memoria**.

Además, **es conveniente hardware adicional para acelerar la traducción** (TLB, MMU), ya que **cada referencia a memoria implica dos accesos a memoria física**. Se requiere un **registro apuntador a la base de la tabla de páginas** (PTBR) del programa o a la tabla de segmentos (STBR). El cálculo de la dirección física es más simple en paginación, ya que sólo requiere la concatenación del número de marco y el desplazamiento, mientras que la segmentación implica una suma de la dirección base del segmento y el desplazamiento.

Para **protección**, en la paginación se puede asociar un conjunto de bits a cada entrada de la tabla de páginas:

- bit de sólo lectura,
- bits para restricciones de acceso (en sistemas multiusuario).

Además del soporte hardware para la traducción de direcciones de los sistemas paginados, **la memoria virtual requiere mecanismos hardware adicionales**:

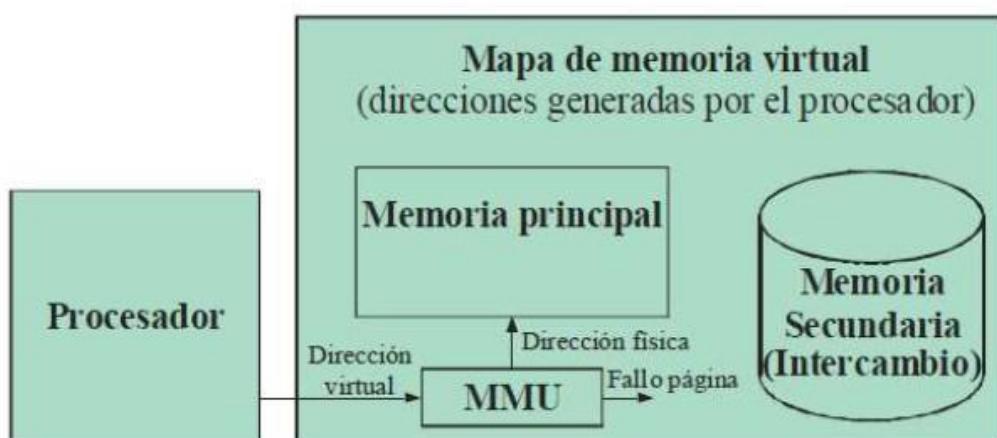
- Espacio para paginación en un dispositivo de almacenamiento secundario (disco).

- Bit de validez, V. Para cada entrada de la tabla de páginas es necesario un bit que indique si la página correspondiente está cargada en memoria o no.
- Trap de fallo de página. Cuando la página referenciada no está cargada en memoria, el mecanismo de interrupciones produce el salto a la rutina de tratamiento del fallo de página (que promoverá la carga de la página en memoria). A diferencia de una interrupción normal, el fallo de página puede ocurrir en cualquier referencia a memoria durante la ejecución de la instrucción, por lo que la arquitectura debe proporcionar los mecanismos adecuados para establecer un estado del procesador consistente antes de saltar a la rutina de tratamiento.
- Información adicional para la gestión del fallo de página (bit de página modificada, referenciada, ...)

## Itinerario 11: Gestión de Memoria Virtual

### Gestión de Memoria virtual

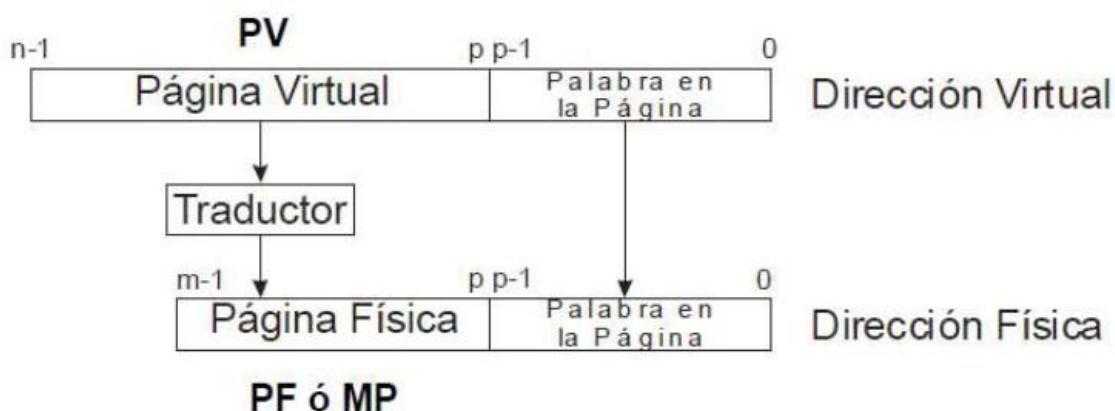
- Gestión automática de la parte de la Jerarquía de Memoria formada por los niveles de memoria principal y de disco.



## Beneficios

- Beneficios:
  - Protección y reubicación de procesos
  - Aumento grado de multiprogramación
    - ¡Cuidado con “hiperpaginación” [trashing]!
  - Ejecución de programas que no caben en M. ppal.
- El proceso sólo ve **direcciones virtuales**.
- El mapa virtual de un proceso está soportado en memoria secundaria (*swap*).
- No es necesario que todo el proceso esté cargado en memoria =>
  - Más procesos en MP, uso más eficiente del procesador.
  - Un proceso puede ser mayor que la MP.
  - Arranque más rápido.
- **Conjunto residente:** parte del proceso cargado en MP.

## Paginación: Mecanismo de traducción



- Necesario una Tabla de Páginas, TP.

## Ventajas de segmentación

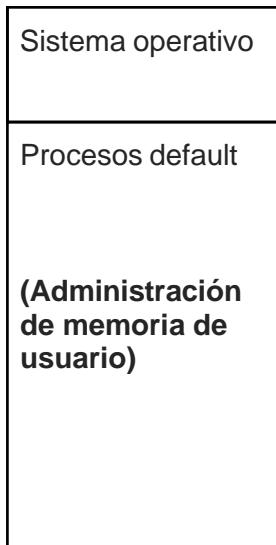
- Dos procesos pueden compartir un segmento con solo tener entradas en sus tablas generales que apunten al mismo segmento del almacenamiento primario.
- En un sistema de segmentación, una vez que un segmento ha sido declarado como compartido, entonces las estructuras que lo integran pueden cambiar de tamaño.

**YouTube: Segmentación de la memoria RAM y virtual**

(mirar video a partir de **2:20** para ver mapeo)

## VideoConferencia 16/10/2022

Memoria RAM

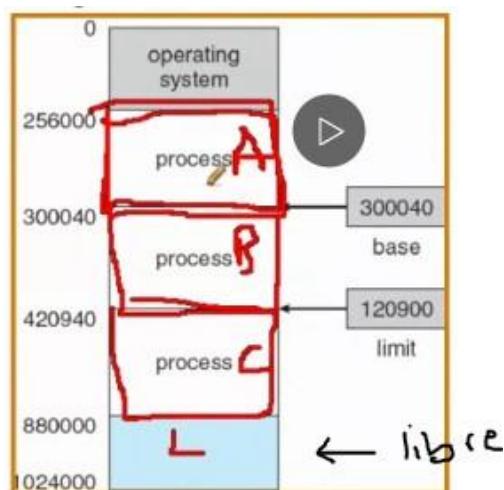


Para la parte de administración de memoria de usuarios hay **2 grandes grupos**:

- Administración de memoria contigua.
- Administración de memoria no contigua

### Administración de memoria contigua

Cuando se administra la memoria en forma **contigua** todo el proceso tiene que estar almacenado en direcciones contigua de memoria



Proceso A Almacenado en direcciones **contigua** de memoria

Proceso B Almacenado en direcciones **contigua** de memoria

Proceso C Almacenado en direcciones **contigua** de memoria

Se llama **CONTIGUA** porque el proceso que se guarda en la memoria principal se le asigna un conjunto de direcciones de memoria (**Partición**) contiguas.

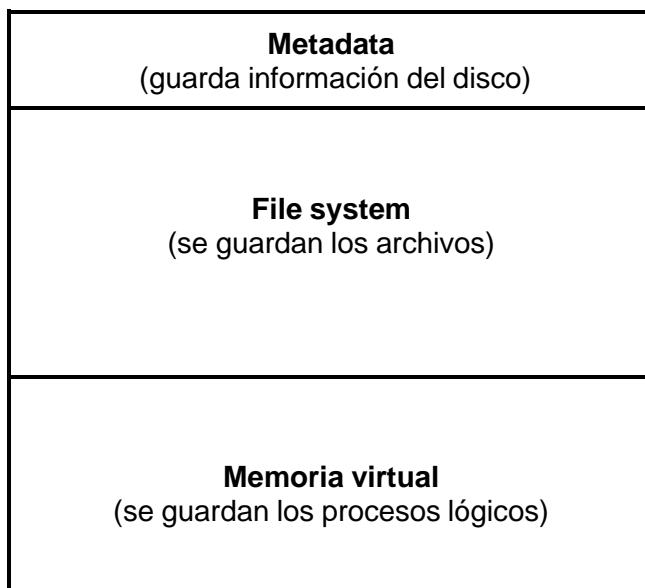
Si entra algún proceso que entra en la partición **libre**, será asignada esa partición, si no entra, será rechazada

## Dirección lógica y dirección física

**Dirección lógica** - generada por el CPU; también conocida como **dirección virtual**.  
**Siempre relativo a 0.**

**Dirección física** - dirección vista por la unidad de memoria.

Disco



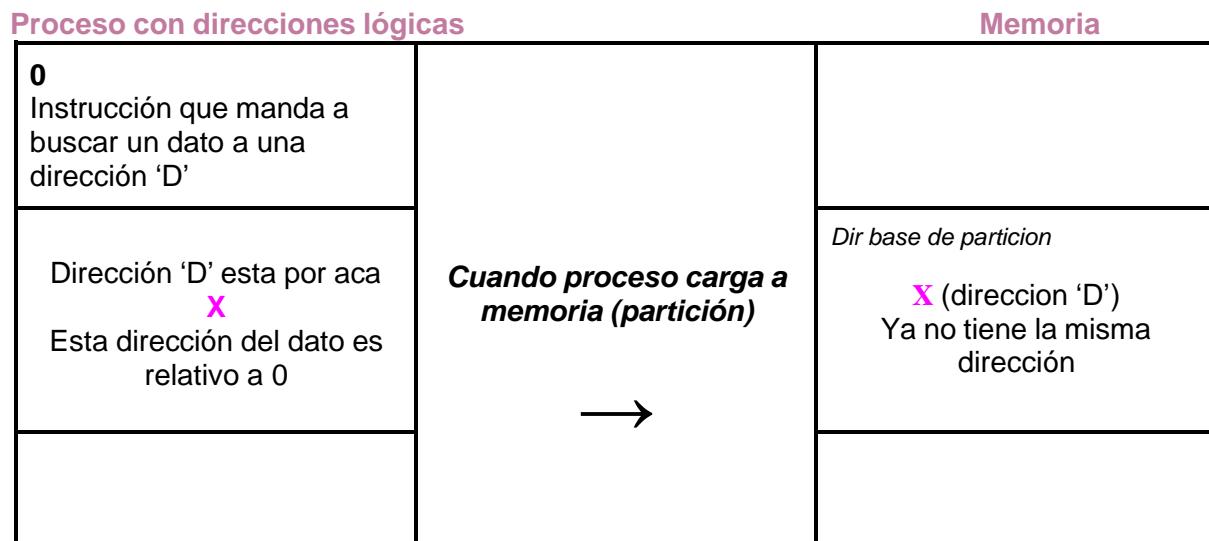
Cuando un proceso va de estado nuevo a estado listo, se guarda en la dirección lógica. Las direcciones lógicas siempre empiezan de 0.

Si hay 2 procesos en estado listo, los dos tendrán como comienzo, la dirección lógica 0. Tienen la misma dirección, **lógicamente serían lo mismo pero físicamente no**.

Una vez que carga el proceso en la memoria RAM (físico) ya no estará en la dirección 0, sino, donde sea que asigna.

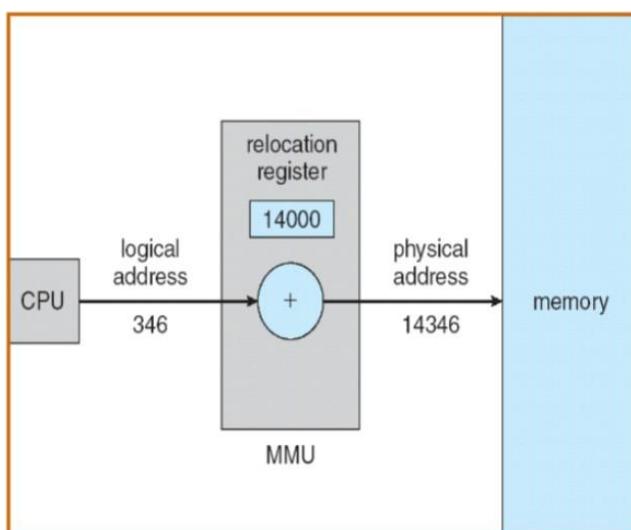
## MMU (Unidad de Manejo de Memoria)

**Mapea** direcciones virtuales / lógicas a físicas. Se encarga de convertir la dirección lógica a física. El proceso en su ejecución genera direcciones lógicas.



Lo que hace la MMU es que cuando el proceso genera la dirección **X** tiene que obtener la dirección física **X**.

El valor en el registro de relocalización **se añade** a cada dirección generada por un proceso de usuario al momento de enviarla a la memoria.



Acá vemos que el proceso genera una dirección lógica (**346**)  
La dirección física es **14346**

La dirección **base** donde comienza la partición donde está almacenado el proceso de usuario es **14000**. Entonces, donde comienza el proceso en memoria física (**dirección 14000**) le sumó la dirección lógica relativa a 0 (**346**) obtenemos la dirección física **14346**.

En otras palabras; *para obtener la dirección física, la MMU le suma a la dirección lógica la base donde comienza la partición que almacena el proceso.*

## Asignación contigua

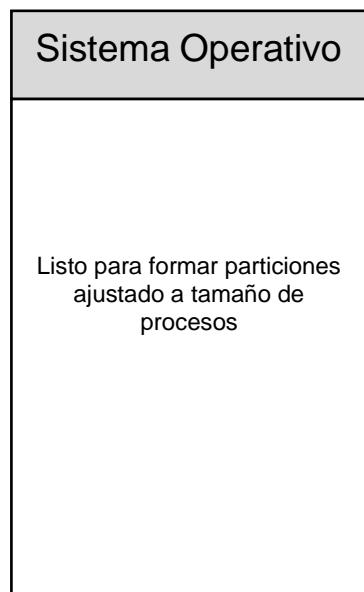
- Las direcciones generadas por los procesos tienen que ser **>= (mayor || igual)** a la base de la partición que le corresponde al proceso.
- Tiene que ser **< (menor)** al límite de la partición que le corresponde al proceso.
- Esto lo mapea el **MMU** dinámicamente.

### Tipos de administración contigua:

- Administración de memoria **contigua con particiones fijas** con:
  1. Particiones de igual tamaño
  2. Particiones de diferente tamaño
- Administración de memoria particionada fija variable.

## Administración de memoria particionada fija variable

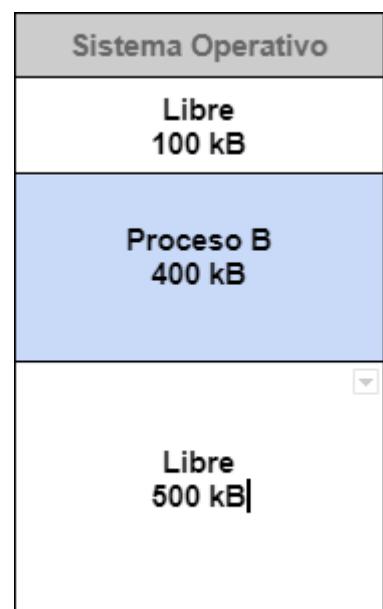
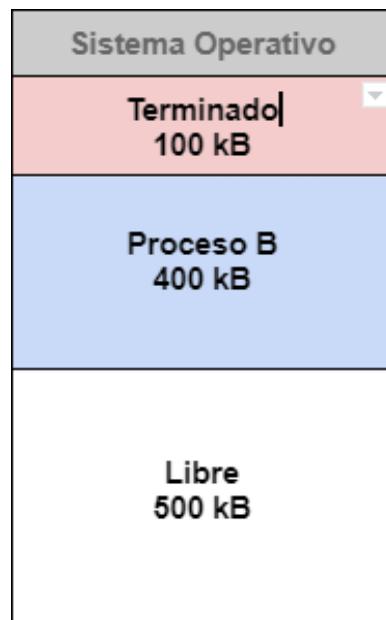
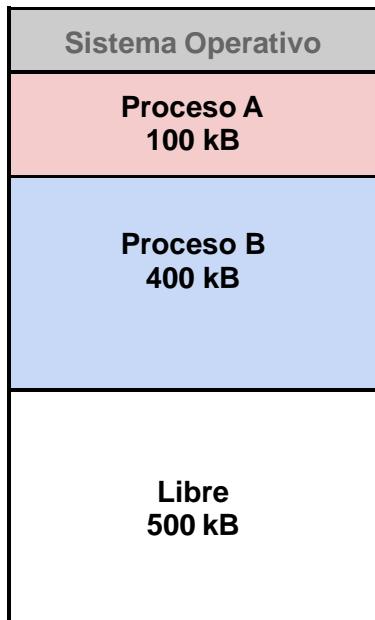
Cuando arranca la sesión, las particiones se van formando en forma dinámica a medida que se cargan los procesos en memoria.



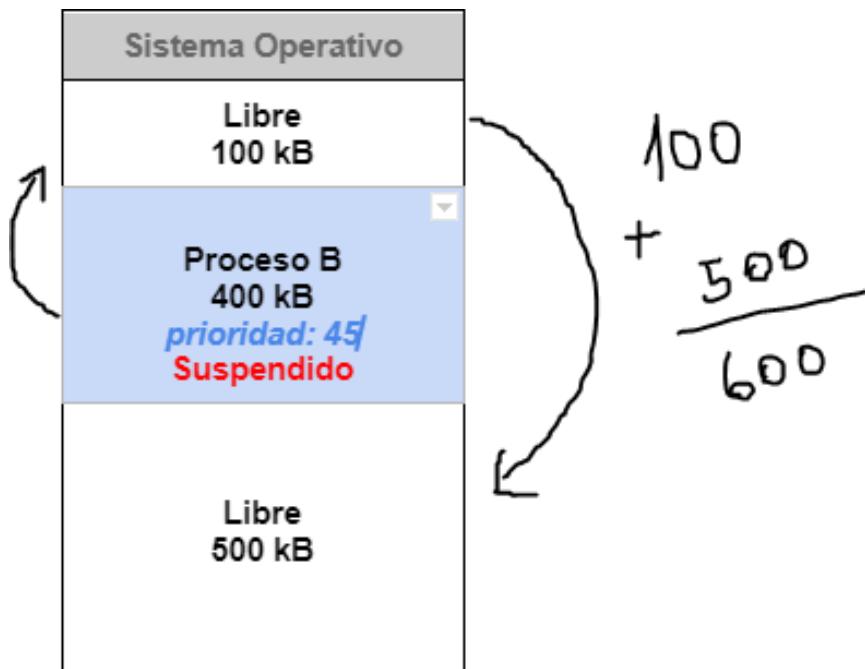
Las particiones se van **generando dinámicamente**, se van **ajustando exactamente al tamaño del proceso**, puede haber **reubicación** y **compactación** siempre y cuando el proceso que quiere entrar a memoria es de **muy alta prioridad**.

**No** tiene sentido suspender un programa de **alta prioridad** para reubicarlo y poder meter a un programa de **baja prioridad**.

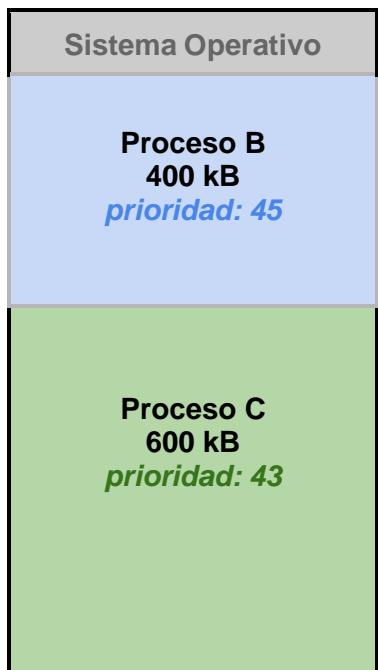
Ejemplo abajo:



#### Nuevo proceso



Así queda la memoria después de mover el hueco de **100 kB** y sumarlo a la partición libre de **500 kB**.



## Problemas de asignación almacenamiento

El mejor ajuste (**Best-Fit**) se usa en la partición fija

El peor ajuste (**Worst-Fit**) se usa en la partición variable, por que no va a tener fragmentación interna, solo externa.

## Administración de memoria NO contigua

Permite ejecutar un proceso que esté almacenado en forma no contigua, partes de procesos pueden estar en diferentes bloques de la memoria física. El proceso se parte en pedazos y cada pedazo se llama página.

Administración de memoria no contigua permite guardar un proceso en estado listo para que tome la CPU en memoria no contigua.

## Paginación

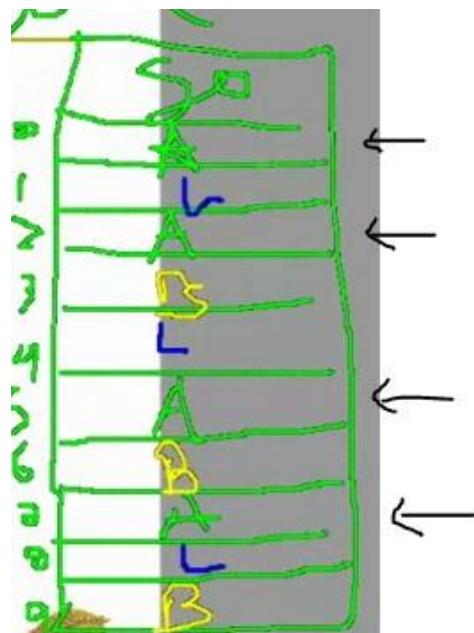
La paginación divide la memoria física en pequeños bloques de tamaños iguales. Sus tamaños oscilan entre 512 bytes y 8 kB. Estos bloques se llaman **frames** o **marcos**.

Construye una tabla de páginas para cada proceso

Proceso A

Tabla-Pagina-A

TP A	
Página	Frame
0	2
1	7
2	5
3	0
etc...	etc...



Esto se lo llama paginación **simple** o **pura** (todo el proceso tiene que estar almacenado en memoria, pero no necesariamente en memoria contigua).

Puede haber fragmentación interna en el frame que contiene la última página del proceso

*Desventaja de no contigua es que las tablas de paginación de los procesos se guardan en la memoria principal y eso es un problema muy grave. Los procesos grandes tienen tablas de paginación muy grandes.*

### Paginación bajo demanda

Permite que un proceso se ejecute sin que esté TODO en memoria.

*Gran ventaja de no contigua es que no hay que compactar las particiones libres para meter un proceso por que los procesos se parten en pedazos y se guardan en frames no contiguas*

## TRABAJO PRÁCTICO 2 OBLIGATORIO

- 1) La gran diferencia entre la administración de memoria contigua y la administración de memoria no contigua es que cuando se administra la memoria en forma contigua, todo el proceso tiene que estar almacenado en direcciones contiguas de memoria.

La administración de memoria no contigua permite ejecutar un proceso que esté almacenado en forma no contigua, partes de procesos pueden estar en diferentes bloques de la memoria física manejados por una tabla de paginación. El proceso se parte en pedazos y cada pedazo se llama página. Además, se puede guardar un proceso en estado listo para que tome la CPU en memoria no contigua.

**Contigua con particiones fijas de 500 kB**

Sistema Operativo
<b>Proceso A (350 kB)</b> Fragmentacion interna: (220 kB)
<b>Proceso B (190 kB)</b> Fragmentacion interna: (220 kB)
<b>Proceso C (280 kB)</b> Fragmentacion interna: (220 kB)
<b>Libre (500 kB)</b> <i>Fragmentacion externa</i>

**No contigua paginación (512 bytes)**

Frame	Sistema Operativo
0	Proceso B
1	Proceso C
2	Libre
3	Proceso A
4	Libre
5	Proceso B
6	Proceso A
7	Proceso A
8	Libre
9	Proceso A

2) Las administraciones de memoria **contigua** son:

- Particiones fijas con particiones de igual tamaño
- Particiones fijas con particiones de distinto tamaño
- Particiones variables

Las administraciones de memoria **no contigua** son:

- Paginación simple
- Paginación bajo demanda
- Segmentación

3) En **memoria contigua de particiones fijas con particiones de igual tamaño** el proceso que se guarda en la memoria principal se le asigna un conjunto de direcciones de memoria (**Partición**) contiguas.

Si entra algún proceso que entra en la partición **libre**, será asignada esa partición, si no entra, será rechazada. Si el tamaño del proceso es más pequeño que la partición, esta partición tendrá fragmentación interna porque el hueco no podrá ser usado por ningún otro proceso.

En **memoria contigua de particiones fijas con particiones de distinto tamaño** las particiones son de diferentes tamaños y el proceso se guarda en particiones dependiendo del tamaño de partición y proceso.

En **memoria contigua particionada fija variable**, cuando arranca la sesión, las particiones se van formando en forma dinámica a medida que se cargan los procesos en memoria.

Las particiones se van generando dinámicamente, se van ajustando exactamente al tamaño del proceso, puede haber reubicación y compactación siempre y cuando el proceso que quiere entrar a memoria es de muy alta prioridad.

En **memoria no contigua paginación simple**, la paginación divide la memoria física en pequeños bloques de tamaños iguales. Sus tamaños oscilan entre 512 bytes y 8 kB. Estos bloques se llaman **frames** o **marcos**. Construye una tabla de páginas que se guarda en memoria principal para cada proceso y todo el proceso tiene que estar almacenado en memoria, pero no necesariamente en memoria contigua.

Puede haber fragmentación interna en el frame que contiene la **última página** del proceso no más.

En **memoria no contigua paginación bajo demanda**, se permite que un proceso se ejecute sin que esté TODO en memoria.

En **memoria no contigua segmentación**, es un esquema de manejo de memoria mediante el cual la estructura del programa refleja su división lógica. Divide el programa en sus unidades lógicas (código, pila, datos, ...), denominadas **segmentos**.

En la **tabla de segmentos**, tiene el número de segmentos (como la tabla de páginas) que son de distintos tamaños. Los segmentos contienen las direcciones donde comienza su segmento y cuanto ocupa cada segmento.

#### 4) Mapeo de dirección lógica a dirección física en las administraciones de memoria contigua.

### MMU (Unidad de Manejo de Memoria)

**Mapea** direcciones virtuales / lógicas a físicas. Se encarga de convertir la dirección lógica a física.

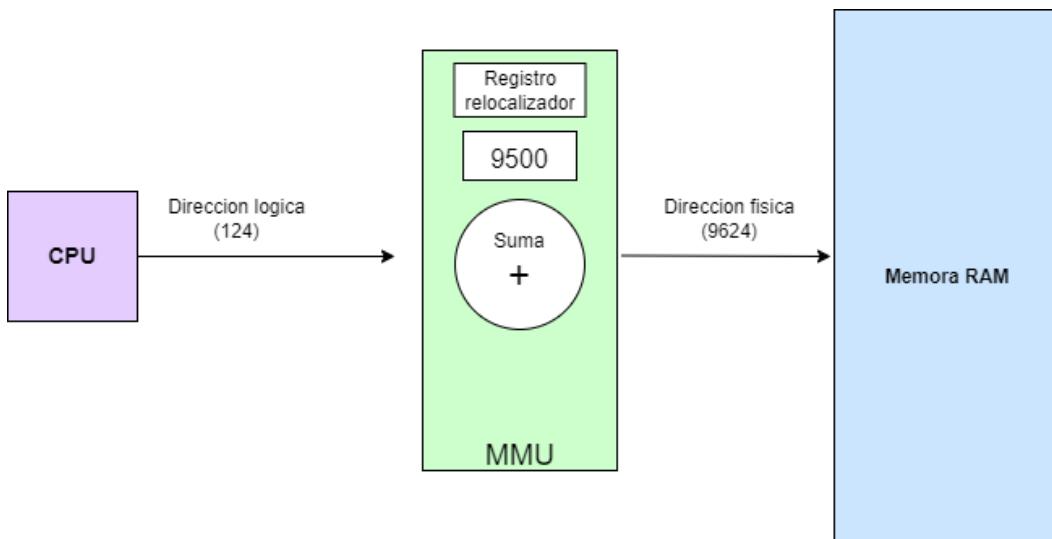
Cuando un proceso va de estado nuevo a estado listo, se guarda en la dirección lógica. Las direcciones lógicas siempre empiezan de 0.

Una vez que carga el proceso en la memoria RAM (físico) ya no estará en la dirección 0, sino, donde sea que el **MMU** lo asigna.

Proceso con direcciones lógicas		Memoria
<b>0</b> Instrucción que manda a buscar un dato a una dirección 'N'		
Dirección 'N' esta por aca <b>X</b> Esta dirección del dato es relativo a 0	<b>Cuando proceso carga a memoria (partición)</b>	<i>Dir base de particion</i> <b>X</b> (dirección 'N') Ya no tiene la misma dirección
	→	

Lo que hace la MMU es que cuando el proceso genera la dirección **X** tiene que obtener la dirección física **X**.

El valor en el registro de relocalización **se añade** a cada dirección generada por un proceso de usuario al momento de enviarla a la memoria.



Acá vemos que el proceso genera una dirección lógica (**124**)  
La dirección física es **9624**

La dirección **base** donde comienza la partición donde está almacenado el proceso de usuario es **9500**. Entonces, donde comienza el proceso en memoria física (**dirección 9500**) le suma la dirección lógica relativa a 0 (**124**) obteniendo la dirección física **9624**.

## 5) Mapeo de dirección lógica a dirección física en las administraciones de memoria no contigua.

La paginación divide la memoria física en pequeños bloques de tamaños iguales. Sus tamaños oscilan entre 512 bytes y 8 kB. Estos bloques se llaman **frames** o **marcos**.

El proceso al ingresar a la memoria se parte en pedazos pequeños llamados páginas, del tamaño de los frames y se guardan en estos mismos frames. Se construye una tabla de páginas para cada proceso y se guardan en la memoria principal.

Si llega un proceso A, se crea una tabla de páginas específica para el **proceso A** donde indica que página de proceso está en cual frame de la memoria.

The diagram illustrates the mapping between a process page table (TPA) and a memory frame table. On the left, the TPA table has columns for Page and Frame. On the right, the frame table has columns for Frame and Sistema Operativo. Arrows point from each row in the TPA to its corresponding entry in the frame table.

TPA	
Página	Frame
0	5
1	3
2	4
3	1
4	8
etc...	etc...

Frames	Sistema Operativo
0	Otro proceso
1	Proceso A
2	Otro proceso
3	Proceso A
4	Proceso A
5	Proceso A
6	Otro proceso
7	Libre
8	Proceso A

Esto se lo llama paginación **simple** o **pura** (todo el proceso tiene que estar almacenado en memoria, pero no necesariamente en memoria contigua).

#### 6) Ventajas y Desventajas de contigua y no contiguas:

Contigua		No contigua	
Ventajas	Desventajas	Ventajas	Desventajas
Tiende a ser más rápido de ejecutar	Fragmentación interna en casi todas las particiones en administración de partición fija si el proceso del usuario es de menor tamaño que el tamaño de partición.  <i>Huecos</i>	No hay que compactar las particiones libres para meter un proceso porque los procesos se parten en pedazos y se guardan en frames no contiguas	las tablas de paginación de los procesos se guardan en la memoria principal y eso es un problema muy grave.  <i>Los procesos grandes tienen tablas de paginación muy grandes.</i>
Más fácil de manejar para el sistema operativo.	No se puede cambiar el tamaño de las particiones en tiempo de ejecución.  <i>(van a ser fijos durante toda la sesión del sistema).</i>	Dos procesos pueden compartir un segmento con solo tener entradas en sus tablas generales que apunten al mismo segmento del almacenamiento primario	Los apuntadores de la estructura de lista consumen espacio en disco.

Menos gastos generales.	Fragmentación externa si la partición no fue usada por que el proceso no entra.	En un sistema de segmentación, una vez que un segmento ha sido declarado como compartido, entonces las estructuras que lo integran pueden cambiar de tamaño.	Debido a la posible dispersión en el disco, la recuperación de registros lógicamente contiguos puede significar largas búsquedas.
Resolución de direcciones en tiempo de carga.	Baja utilización de memoria.	Admite carga dinámica y vinculación dinámica.	Duplica el tiempo efectivo de acceso a memoria. <i>Paginación</i>
Cada proceso puede direccionar el total de memoria disponible. <i>Particion variable</i>	Necesidad de realizar compactación. <i>Particionada tamaño distinto</i>		Dirección lógica no pertenece al rango de tamaño del segmento. El proceso termina. <i>Segmentacion</i>

## VideoConferencia 30/10/22

¿Qué sucede si no hay frames libres?

### ¿Qué sucede si no hay frames libres?

- Reemplazo de página – encuentra una página en memoria, que NO está en uso y sácala
  - algoritmo
  - rendimiento – queremos algoritmo que nos de el menor número de faltas de página
- Misma página puede ser traída a memoria muchas veces

### Reemplazo de página

Ahora agregamos un bit más (*bit de modificación / dirty bit*), bit de invalidez o bit de validez, cada página / frame tiene asociado un bit que indica si la página que está alojada en el frame, sufrió una modificación. Esto es clásico en las páginas que contienen datos, o datos estáticos o datos dinámicos por que el código de un programa no se puede modificar. Lo que sí se puede modificar son los datos del proceso.

¿Qué sucede con las páginas que se modifican? Cuando se le modifica se le pone

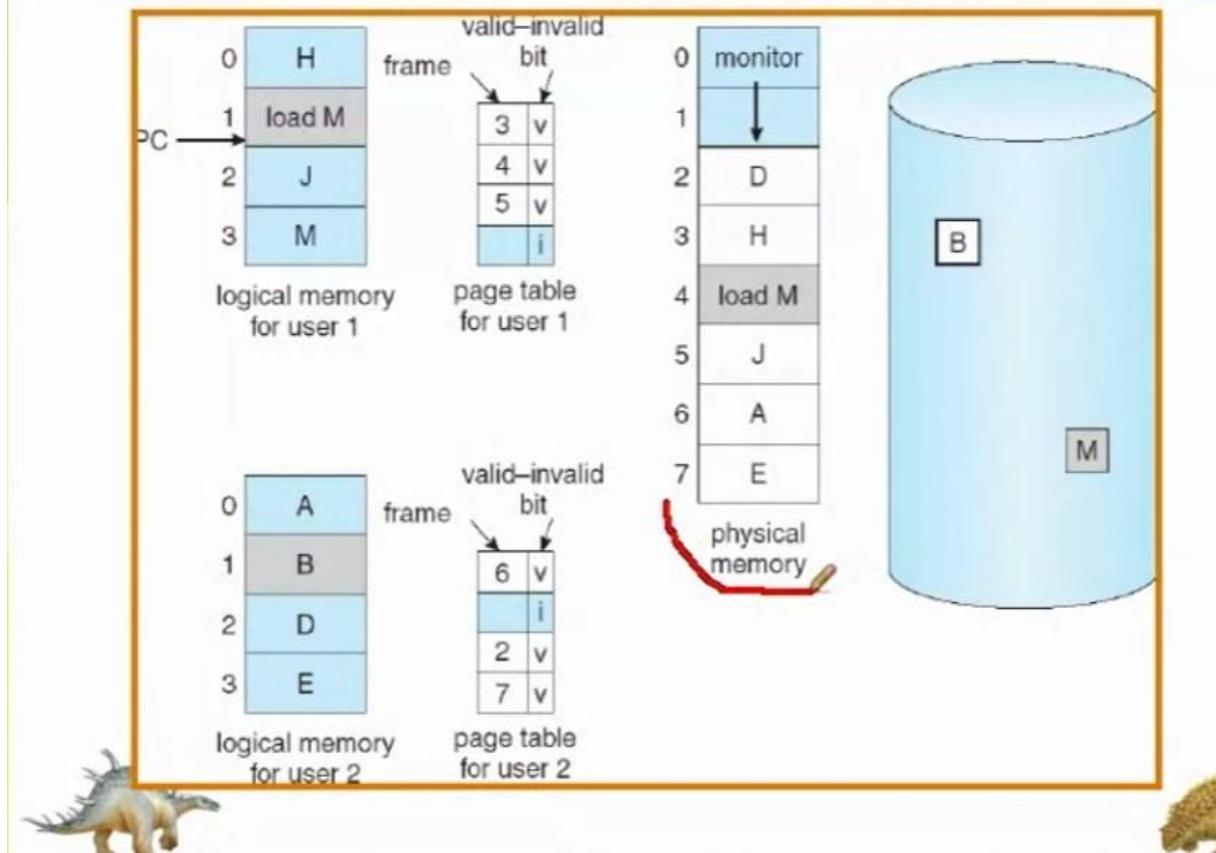
un valor bit para que los algoritmos de reemplazo de páginas no seleccionen páginas que se han modificados, por qué si la página fue modificada, y se selecciona como víctima, no se puede pisar esa página. Cuando es reemplazada la pisas, pero si fue modificada, antes de pisarla hay que hacer una entrada y salida en disco por que a la pagina la tengo que copiar en el disco.

Las entradas y salidas sobre el disco de almacenamiento magnético perjudican al rendimiento del proceso, porque casualmente la entrada y salida es muy lenta con respecto a un acceso de memoria, perjudicando el rendimiento del proceso.

Podemos ejecutar procesos que superan en tamaño el tamaño de la memoria real física, por que lo podemos ir ejecutando por partes. Además, si la memoria es pequeña y está llena, y los procesos son muy grandes, entonces ahí tenemos el reemplazo de páginas, reemplazamos una página para poner la otra.

(mirar desde **7:40 a 11:20** para explicación visual de reemplazo de página o mi explicación de eso abajo)

## Reemplazo de página necesario



**Memoria física - Lleno (todos los frames ocupados)**

**Tabla de página del proceso 1** - Tiene páginas que no están cargadas en la memoria (el cuadrito azul es la **página 3** que no tiene un frame asignado)

**Tabla de página del proceso 2** - Tiene páginas que no están cargadas en la memoria (el cuadrito azul es la **página 1** que no tiene un frame asignado)

**Memoria lógica para proceso 2** - La página 1 representa la '**B**' en la memoria virtual

**Memoria lógica para proceso 1** - La página 3 representa la '**M**' en la memoria virtual

*Las páginas que están cargadas en memoria también están en memoria virtual, estarán todas las páginas de los dos procesos también en la memoria virtual.*

Se está ejecutando la instrucción cargue el **dato M** (página 1 **proceso 1**)

**Dato M** genera una referencia (dirección lógica) a la página 3 ('**M**') del mismo proceso por que '**M**' está en la página 3.

Se accede a la tabla de páginas indexada por número de página y dice "**la página 3 no está cargada en memoria**" como vemos que tiene el bit de **invalid**

*Ahora hay que ejecutar un **algoritmo de reemplazo de páginas**.*

El algoritmo de reemplazo de páginas puede ser local o puede ser global.

Si es **local**, va a buscar como víctima una página del mismo proceso que generó el fallo de página (**proceso 1**, pagina 0, página 1 o pagina 2)

Un reemplazo **global**, podría elegir entre la pagina 0, pagina 1 o pagina 2 del proceso 1, pero tambien podria elegir entre la pagina 0, pagina 2 o pagina 3 del **proceso 2**.

### Reemplazo de página básico

## Reemplazo de página básico

1. Encontrar la localidad de la página deseada en disco
2. Encontrar un frame libre:
  - i. Si existe un frame libre, úsalo
  - ii. Si no hay frame libre, utiliza **algoritmo de reemplazo** para seleccionar frame **victima**
3. Traer la página deseada al (recién creado) frame libre; actualiza las tablas de páginas y frame
4. Re-inicia el proceso

El punto 4 de la captura de arriba es ‘**Re-inicia la instrucción (NO EL PROCESO)**’

## Algoritmos de reemplazo de páginas

### Algoritmo FIFO (First-In First-Out)

Este algoritmo usa la **selección local** para reemplazar páginas.

Va a elegir como víctima una página del mismo proceso que se generó fallo de página. El FIFO va a ver cual de las páginas del proceso entró primero y lo asigna como víctima a ese para reemplazarlo por la nueva página. Luego para la siguiente página si no hay espacio, busca el siguiente al que fue reemplazado posteriormente para ser reemplazado por el nuevo, y así sucesivamente hasta terminar el proceso

**(ejemplo visual)**

ALGORITMO FIRST IN FIRST OUT														
PROCESO A	1	2	3	4	1	2	5	1	2	3	4	5		
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5		
Frame 2		2	2	2	1	1	1	1	1	3	3	3		
Frame 3			3	3	3	2	2	2	2	2	4	4		
Fallo de página	x	x	x	x	x	x	x			x	x			9 fallos

Básicamente si la página en específico del proceso no está en ningún frame (como el ejemplo que está resaltado), causa un fallo de página (la ‘x’) y se reemplaza por la primera pagina que entro.

Como vemos en la primera columna marcada en rojo, la memoria tiene todos sus frames ocupados con las páginas 4, 2 y 3 del **proceso A**, luego en la segunda columna, la siguiente instrucción que viene es la página 1 del **proceso A**, entonces reemplaza la página 2 que está en frame 2 por qué de esas 3 paginas (4, 2 y 3) la página 2 llegó primero.

Así se va reemplazando sucesivamente hasta terminar el proceso.

### Anomalia de Belady

Más frames → más faltas de páginas

**Algoritmo Óptimo (mira el futuro en vez de el pasado)**

ALGORITMO OPTIMO														
PROCESO A	1	2	3	4	1	2	5	1	2	3	4	5		
Frame 1	1	1	1	1	1	1	1	1	1	1	4	4		
Frame 2		2	2	2	2	2	2	2	2	2	2	2		
Frame 3			3	3	3	3	3	3	3	3	3	3		
Frame 4				4	4	4	5	5	5	5	5	5		
Fallo de página	x	x	x	x			x				x			6 fallos

Como se ve la parte resaltada del **proceso A**, el proceso solicita un frame para la página 5, entonces mira al futuro para ver cuales son las siguientes páginas. Como la página 4 es la más lejana, se reemplaza esa página del frame 4 como vemos en la columna resaltado en rojo y nos quedamos con páginas 1, 2, 3 y 5 (siguiente columna).

Este algoritmo no existe por que nadie y nada puede ver el futuro, por eso, simplemente usamos este para comparar la cantidad de fallo de páginas cuando creamos un algoritmo.

*La cantidad de fallo de páginas del algoritmo que diseñamos debe acercarse a la cantidad de fallo que tiene el algoritmo óptimo.*

Algoritmo Menos Recientemente Utilizado (LRU) → Less recently used

ALGORITMO LRU														
PROCESO A	1	2	3	4	1	2	5	1	2	3	4	5		
Frame 1	1	1	1	1	1	1	1	1	1	1	1	5		
Frame 2		2	2	2	2	2	2	2	2	2	2	2		
Frame 3			3	3	3	3	5	5	5	5	4	4		
Frame 4				4	4	4	4	4	4	3	3	3		
Fallo de página	x	x	x	x			x			x	x	x		8 fallos

Aca vemos en la fila resaltado en rojo que la página menos recientemente usada es la número 3, entonces como vemos en la segunda columna resaltado en rojo, cuando el **proceso** causa fallo de página y solicita la página 5, la página 3 es

reemplazado por la 5. Así se van reemplazando hasta que termine el proceso.

### Cómo implementar este algoritmo?

Implementación con contador

- 
- Cada entrada de página tiene un contador; cada vez que es referida, copia el reloj al contador
  - Cuando una página requiere ser cambiada, revisa contadores para determinar cual cambiar

**Algoritmos de reemplazo de páginas aparecen con administración de memoria paginada bajo demanda y son necesarios para que podamos ejecutar procesos que superan el tamaño de la memoria física cuando la memoria física está completa**

Algoritmo de **segunda oportunidad** (o Reemplazo-Página)

Utiliza un bit de referencia

**(Mirar vc desde 41:30 a 45:55 para ver explicacion del algoritmo)**

## Unidad 4: Gestión de almacenamiento de información

### Métodos de asignación en disco magnético

La asignación de espacios en memoria se refería a los procesos. Métodos de asignación en disco se refiere al tema **archivos (File System)**.

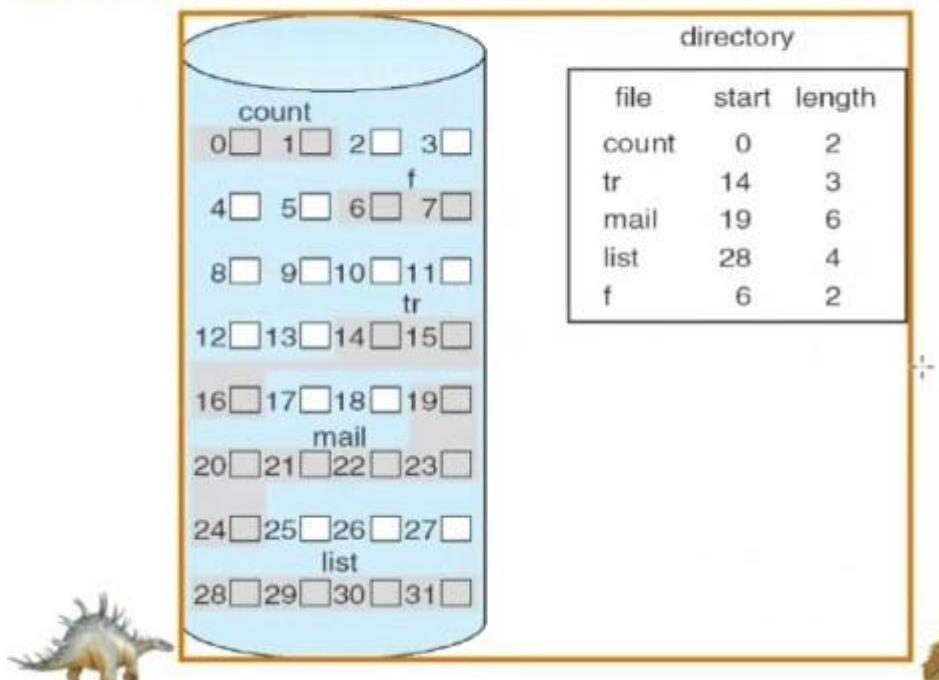
El sistema operativo tiene que tener un método de asignación y un método también de administración de espacio libre.

#### Hay 3 métodos de asignaciones

- **Asignación contigua:** se usa en los mainframe
- **Asignación ligada (o enlazada):** Pequeñas, medianas y grandes sistemas de cómputos.
- **Asignación indexada (o indizada):** Pequeñas, medianas y grandes sistemas de cómputos.

#### Asignación contigua

### Asignación contigua de espacio en disco



El cilindro azul representa un disco de almacenamiento magnético (**área de File System, no área virtual**).

La asignación de espacio de disco también necesita estructura de datos para administrar.

Una estructura es el **Directory**. En asignación de memoria contigua, cuando se asigna espacio un archivo, ese espacio tiene que ser contiguo.

Los cuadraditos son bloques del disco o sectores o registros físicos. En el directory vemos que el archivo **count** empieza en bloque 0 y contiene 2 bloques, el **tr** empieza en bloque 14 y contiene 3 bloques... etc.

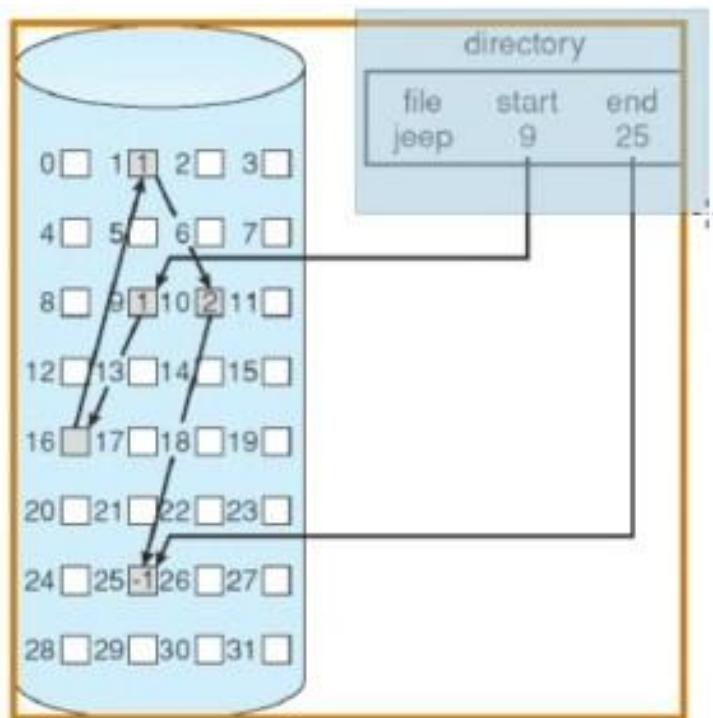
El **problema** con esto es que si el archivo **count** quiere crecer y necesita 5 bloques más, no tendrá ese espacio solicitado porque estaría chocando con el archivo **f**.

La **ventaja** que tiene es el acceso directo. Se puede recorrer un archivo muy rápidamente en forma secuencial y también podes encontrar muy rápidamente un registro del archivo en forma directa.

## VideoConferencia 06/11/22

### Asignación ligada

Se hace de la misma manera que se implementa una lista simplemente enlazada en memoria dinámica.



Nombre del archivo - **jeep**

Dirección / número de bloque / registro físico de comienzo del archivo - **9**

Dirección / número de bloque / registro físico de donde termina el archivo - **25**

**Bloque 9** - comienza el archivo, tendrá dos tipos de información: **Datos del archivo** e **información de gestión**.

**Información de gestión** - El número de bloque que continúa en la lista.

Para recorrer el archivo, **solo** se podrá recorrer de forma secuencial.

La **ventaja** de asignación ligada es que puede tener bloques libres que están distribuidos en cualquier lugar del dispositivo, no necesariamente tienen que ser bloques contiguos.

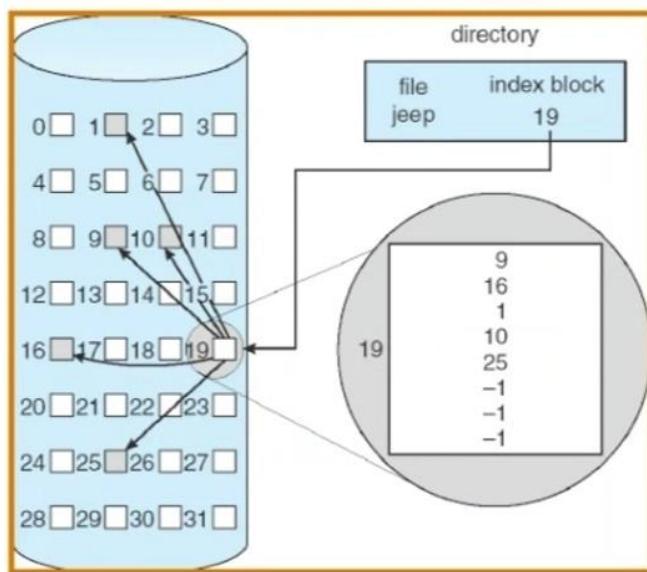
Fácil de asignar espacio al archivo en el disco

\***ACLARACION\*** En todas las administraciones de espacio existe una administración de espacio paralela que es la administración de **espacio libre**.

La **desventaja** es que no se puede hacer un acceso directo, no tenemos una expresión algebraica que dado un número de registro lógico acceda directamente al bloque físico que lo contiene.

La única forma de encontrar un dato dentro de un archivo es hacer una búsqueda secuencial a partir del comienzo (**Bloque 9**)

### Asignación indexada



En el directorio tenemos el **nombre de archivo** y un número de bloque que no apunta a un bloque de datos, sino que **apunta un bloque de índice**.

**Bloque 19** - No contiene datos del archivo, es un registro físico o un bloque de gestión.

Los números que vemos en el zoom del bloque 19 son números de bloque que contiene los datos del archivo.

Quiere decir que el archivo jeep tiene asignado los bloques **9, 16, 1, 10 y 25**.

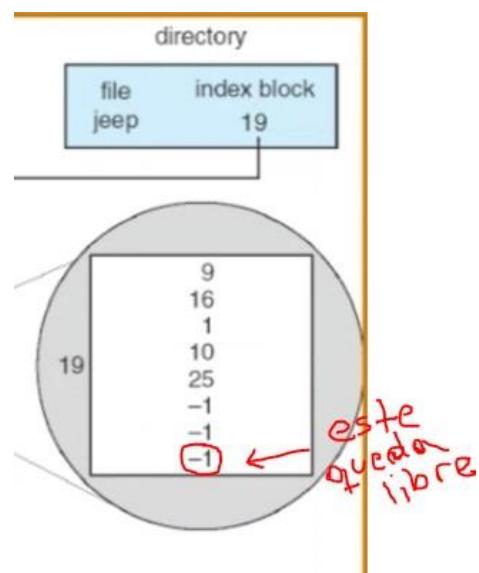
También tiene posibilidades de seguir anotando direcciones de bloques (**los -1**) a medida que el archivo crece.

Si el archivo crece entonces el sistema va a buscar cualquier bloque libre que tenga a partir de la administración de espacio libre, entonces la asignación de espacio le va a pedir a una función que le va a retornar un número de bloque libre. El bloque deja de estar libre para estar ocupado, además indicado a través del directorio (bloque de índices) que ese número pertenece al archivo ‘jeep’.

La **ventaja** de asignación indexada es que podemos asignar cualquier bloque que esté libre y no necesariamente tienen que estar contigua al anterior o siguiente. Podes implementar un acceso directo. Si yo quiero acceder al bloque 1, qué es el tercero lógico del archivo (bloque 19 del archivo jeep).

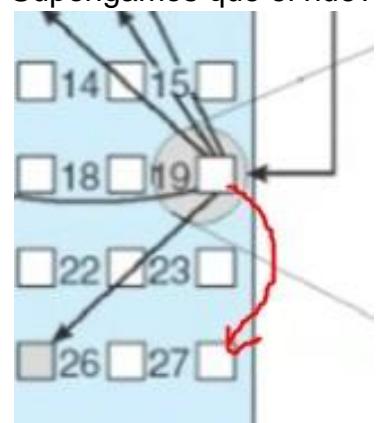
¿Dónde noto ese número de bloque que ya no entra en ese bloque de índice 19?

Lo que hace es reservar el último espacio en el bloque de índice para guardar un número que apunta a otro bloque de índice.



Queda libre para guardar un puntero que te lleva a otro bloque de índice si el archivo ‘jeep’ crece más del límite del tamaño del **bloque de índice 19**.

Supongamos que el nuevo bloque de índice que le sigue al 19 es el 27.



En teoría podemos decir que la asignación ligada está implementada en DOS y Windows.

(mirar desde 19:20 hasta 35:39 para ver explicación de FAT y clusters)

El sistema operativo que tomó la idea conceptual de indexación es **UNIX**.

**Nodo i** - Tiene que ver con el file system de unix. La i es de índice, y el nodo es estructura contigua de datos en el disco.

Directorio de UNIX tiene:

- **Nombre de archivo**
- **Nodo i (numero de nodo)**

## VideoConferencia 13/11/22

Directorio (directorio de unix tienen 2 atributos, los de windows tiene más)	
Nombre	Nodo i
A	12344

12344
<i>Parte 1</i> Número de nodo Tipo de archivo Permisos Cantidad de enlaces Dueño Grupo del dueño Tamaño Fecha de creación Hora de creación Nombre
<i>Parte 2</i> A partir de esta parte ya estos datos (partes del nodo i) guardan la ubicación de los datos del archivo en el dispositivo de almacenamiento magnético.  10 Números del bloque físico del disco que contienen los datos del archivo (Dirección de bloque 764634) este numero apunta a un bloque de datos (Dirección de bloque 345543) este numero apunta a un bloque de datos (Dirección de bloque 116576) este numero apunta a un bloque de datos (Dirección de bloque 986877) este numero apunta a un bloque de datos (Dirección de bloque 087545) este numero apunta a un bloque de datos (Dirección de bloque 242345) este numero apunta a un bloque de datos .... Estos números se los llaman punteros directos.
<i>Parte 3</i> indirecto simple Bloque de índice 62342 —> contiene 4 punteros a bloques de datos En la primera línea de la parte 3 guarda una dirección a un bloque de índice si el archivo crece y demanda más de 10 bloques de datos.
Indirecto doble Puntero 23444 —> apunta a un bloque de índice con 8 punteros, cada uno de los 8 punteros apunta a otro bloque de índices, cual contiene punteros que apuntan a bloques de datos.
Indirecto triple Puntero 43454 —> apunta a un bloque de índice que apunta a bloques de índice donde todos esos indice van a apuntar a bloques de índices que apuntan a bloques de datos

La **diferencia** de esto con la FAT es que partes de la FAT están cargadas en memoria RAM, y ahí la FAT tiene datos o información de asignación de espacio en disco de los archivos, también contiene información sobre espacio libre.

En cambio el **Nodo i de UNIX solo maneja el espacio asignado a un archivo.**

La FAT tiene toda la información de **espacio libre** y **espacio asignado** de todos los archivos. Entonces cuando vos cargas parte de la FAT a memoria, llevas a memoria **información de archivo que quizás no está abierto y ningún proceso lo está usando.**

En cambio, **una importante diferencia** además que el directorio es mucho más pequeño en UNIX y que cualquier **búsqueda secuencial en el directorio de UNIX es más rápida** por que los registros son más pequeños, entonces se guarda el directorio en menores cantidad de bloques de discos.

El **tiempo de entrada y salida** va a ser menor por que tenemos que leer menos bloques de discos para leer un directorio.

**En el caso del Nodo i en UNIX, cuando se levanta un Nodo i a memoria?**

Cuando un proceso hace **OPEN.**

Quiere decir que cuando el proceso OPEN del archivo A, entonces ahí recién se levanta el Nodo i a memoria. O si no, no se levanta. UNIX no tiene en memoria Nodos i que están siendo usados por procesos.

**Cuando se libera de la memoria el Nodo i?**

Cuando el proceso hace **CLOSE** del archivo A.

En cambio, la FAT no es así, la FAT puede tener información de asignación de espacio en disco de **archivos que no están siendo usados.**

**¿Qué pasa si el archivo crece y demanda más de 10 bloques de datos?**

Utilizan la primera entrada de la **parte 3** (Con el nombre '*indirecto simple*').

Guarda una dirección a un **bloque de índice**. El FileSystem de UNIX utiliza un bloque para gestionar bloques de índices. Estos **índices son punteros** que apuntan a bloques de datos.

Supongamos que un bloque de índice permite almacenar 8 punteros a bloques.

Y como en la tabla del nodo i vemos que en el bloque de índice hay 4 punteros de bloques. Esto quiere decir que el Archivo A tiene **14 bloques de datos.**

En la primera versión de UNIX en el bloque de índice se podían guardar no 8, sino **256 punteros a bloques de datos.**

**¿Qué pasa si todos los bloques de datos están ocupados en la parte 2, y también el bloque de índice está completamente ocupado con punteros a bloque de datos, y el archivo sigue creciendo?**

Entonces usamos el Indirecto doble (mirar tabla en parte 3)

Tengo 8 punteros que van a apuntar a 8 bloques de índices donde cada bloque de índice va a apuntar a 8 bloques de datos

**¿Qué pasa si el archivo SIGUE creciendo aún más?**

Entonces usamos el Indirecto triple (mirar tabla en parte 3)

En la primera versión de UNIX, el nodo i contiene **10 punteros a bloques de datos**.  
El indirecto simple apuntaba a **256 bloques de datos**.  
El indirecto doble apuntaba a **65536 bloques de datos** ( $256 * 256$ ).  
El indirecto triple apuntaba a **16777216 bloques de datos** ( $256 * 256 * 256$ ).

La suma de esto nos da el tamaño máximo de un archivo en bloques en la primera versión de UNIX.

## Flujo

El directorio es información permanente dentro del disco de almacenamiento salvo que se borre el archivo.

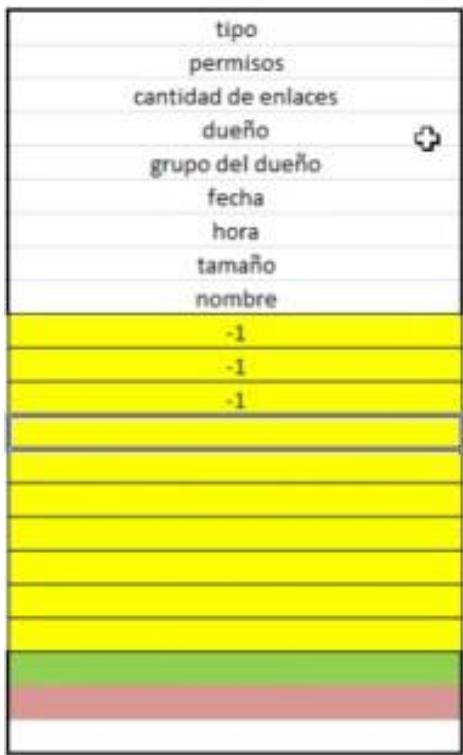
Cuando un proceso crea un archivo 'B' CREAT(B).

Se genera una entrada en el directorio y se le asigna un número de nodo.

Directorio (directorio de unix tienen 2 atributos, los de windows tiene más)	
Nombre	Nodo i
A	12344
B	63533

A partir de ese momento tenemos una entrada en el directorio con el nombre del archivo y el número del nodo asignado y tenemos un nodo i con las 3 partes

**Ver tabla abajo**



La parte 1 con la información del archivo y los bloques de datos en la parte 2 marcados con un -1 para indicar que están libres.

El proceso lo crea pero hace un WRITE(B). Ese WRITE está escribiendo información, entonces el sistema tiene que buscar un bloque libre.

Aca la administración de espacio libre está separada de la administración de espacio ocupado.

Buscará un bloque libre para que apunte a los DATOS.

Cierro el archivo CLOSE(B), el nodo desaparece de memoria (tabla colorida arriba).

El nodo i (tabla) estará en el disco

El día siguiente deseo abrir el archivo OPEN(B) el proceso abre el archivo. Cuando abre el archivo el proceso lo primero que pasa es el sistema va a buscar el nombre asociado al OPEN en el directorio.

Cuando lo encuentra al nombre, encuentra el número de nodo, entonces recien ahí carga el nodo i a memoria.

Pero, los DATOS del archivo B que están en el disco no van a memoria, lo único que ya en memoria es el nodo i cuando hace el OPEN.

¿Cuándo van los datos a memoria?

Cuando el proceso hace READ(B).

Manda a leer el archivo B, entonces ahora si los DATOS van a memoria accediendo a la parte 2 de la tabla.

## Ejercicio (E - Escritura / L - Lectura)

9.5 Consideré un archivo con un tamaño de 100 bloques. Complete el siguiente cuadro indicando cuantas operaciones de E/S se requieren para cada una de las estrategias de asignación si se cumplen las siguientes condiciones:

Condición	Contigua	Enlazada	Indexada
El bloque se agrega al principio			
El bloque se agrega en el medio			
El bloque se agrega al final	▶		
El bloque se borra del principio			
El bloque se borra del medio			
El bloque se borra del final			

Asuma que el bloque de control de archivo y el bloque de índices (en el caso que se use) se encuentran en memoria.

En la asignación contigua asuma que no existe suficiente lugar como para que el archivo crezca hacia el comienzo del mismo, pero sí hacia el final.

Consideré que el bloque de información que se va a agregar se encuentra almacenado en memoria.

Si el bloque se agrega al principio, yo tengo que correr todo el archivo hacia atrás. Entonces tengo que hacer una lectura y escritura por cada bloque por que tengo que leer el último y correrlo a la derecha, leer el ante último y correrlo a la derecha, y así hasta llegar al primer, correrlo a la derecha y después escribir lo que quiero agregar.

*El bloque de índices contiene punteros a bloque de datos y el bloque de índice está en memoria*

En la indexada, los índices están en memoria, lo único que tengo que hacer es desplazar los punteros para hacer espacio para el bloque que yo quiero poner (**1 Escritura**).

Condicion	Contigua	Enlazada	Indexada
El bloque se agrega al principio	$100L + 100E + 1E$	1E	1E
El bloque se agrega en el medio	$50L + 50E + 1E$	$50L + 2E$	1E
El bloque se agrega al final	1E	$100L + 2E$	1E
El bloque se borra del principio	$0L + 0E$ (solo cambias el comienzo del archivo)	1L	$0L + 0E$ (solo desplazas todos los bloques arriba y desaparece el primero)
El bloque se borra del medio	$49E + 49L$	$51L + 1E$	$0L + 0E$ (solo desplazas todos los bloques que están de la mitad inferior hacia arriba 1 bloque y desaparece el del medio)
El bloque se borra del final	$0E + 0L$ (solo cambias la cantidad de bloques del archivo)	$99L + 1E$ (en el bloque 99 pones el EOF)	$0L + 0E$ (al ante ultimo le pones la marca de EOF)

## TRABAJO PRÁCTICO 3 OBLIGATORIO

### Copiar y pegar este código en Visual Studio C# (Ejercicio 1, 2 y 3)

```
//Asignacion Contigua
Console.WriteLine("Asignacion Contigua\n\n");

Console.WriteLine("Ingrese tamaño del Disco Rígido en MB: ");
int DiscoRigidoTamaño = int.Parse(Console.ReadLine()); // ejemplo (1MB)

DiscoRigidoTamaño *= 1048576; //convierto de MB a bytes //(ejemplo: 1 *
1048576 = 1048576 bytes)

Console.WriteLine("Ingrese el tamaño del bloque físico en KB: ");
int BloqueFisicoTamaño = int.Parse(Console.ReadLine()); //tamaño de 1
bloque en KB (ejemplo: 1KB)

BloqueFisicoTamaño *= 1024; //convierto de KB a bytes (ejemplo: 1 * 1024 =
1024 bytes)

Console.WriteLine($"Cantidad de bloques físicos: {DiscoRigidoTamaño /
BloqueFisicoTamaño}"); // (ejemplo: 1048576 / 1024 = 1024 Bloques físicos)

Console.WriteLine("\nIngrese el nombre del archivo: ");
string archnombre = Console.ReadLine();

Console.WriteLine("Ingrese el tamaño del archivo en KB: ");
int archtamaño = int.Parse(Console.ReadLine()); // (ejemplo: 7 KB)

Console.WriteLine($"{"\nIngrese el tamaño de los registros lógicos en
Bytes\n(Preferiblemente menor a {BloqueFisicoTamaño} Bytes): "};
int RegistroLogicoTamaño = int.Parse(Console.ReadLine()); //defino tamaño
de registro lógico (ejemplo: 100 bytes)

int FactorBloqueo = BloqueFisicoTamaño / RegistroLogicoTamaño; //Factor
bloqueo
Console.WriteLine($"Factor bloqueo: {FactorBloqueo}"); // (ejemplo = 10)

int archtamañobytes = archtamaño * 1024;

int CantBloqueArchivo = archtamañobytes / BloqueFisicoTamaño; //al archivo
divido por el tamaño de un bloque físico (ejemplo: 7168 / 1024 = 7 con resto 0)
if ((archtamañobytes % BloqueFisicoTamaño) > 0)
{
    CantBloqueArchivo++; // si la división previa tiene resto mayor a 0, se
    necesita un bloque más para contener el archivo (ejemplo: 7 + 0 = 7 bloques)
}
```

```
Console.WriteLine($"\\n--Directorio--\\nNombre: {archnombre} Bloque inicio: {5}, Cantidad de bloques {CantBloqueArchivo}\\n");

Console.WriteLine("Ingrese el numero de registro logico relativo a 1: ");
int NumeroRegistroLogico = int.Parse(Console.ReadLine()); //elijo un numero de reg logico (ejemplo = 13)

// El registro fisico 5 contiene los registros logicos de 1 a 10
// El registro fisico 6 contiene los registros logicos de 11 a 20 (el registro logico 13 va a estar en este registro fisico)
// El registro fisico 7 contiene los registros logicos de 21 a 30
// El registro fisico 8 contiene los registros logicos de 31 a 40
// El registro fisico 9 contiene los registros logicos de 41 a 50
// El registro fisico 10 contiene los registros logicos de 51 a 60
// El registro fisico 11 contiene los registros logicos de 61 a 70

int RegistroFisicoQueContieneElRegistroLogico = (NumeroRegistroLogico / FactorBloqueo); //La parte entera de la division (ejemplo: 13/10 = 1)
int RegistroFisicoQueContiene =
RegistroFisicoQueContieneElRegistroLogico + 5; //Le sumo la parte entera a la direccion de comienzo (ejemplo: 5 + 1 = 6)
Console.WriteLine($"\\nEl registro fisico que contiene al registro logico: {RegistroFisicoQueContiene}");

int PosRegLogDentroRegFis = NumeroRegistroLogico % FactorBloqueo; // Resto de la misma division de la linea 60 (ejemplo: 13/10 = 1 con resto 3)

Console.WriteLine($"\\nPosicion del registro logico dentro del registro fisico que lo contiene: {PosRegLogDentroRegFis}"); //entonces el registro logico 13 esta en la posicion 3 del registro fisico 6

Console.WriteLine("Presione una tecla para Continuar a Asignacion Enlazada....");
Console.ReadKey();

----- ENLAZADA -----

//Asignacion Enlazada
Console.WriteLine("Asignacion Enlazada\\n\\n");

Console.WriteLine("Ingrese tamaño del Disco Rígido en MB: ");
int DiscoRigidoTamaño1 = int.Parse(Console.ReadLine()); // ejemplo (1MB)

DiscoRigidoTamaño1 *= 1048576; //convierbo de MB a bytes //(ejemplo: 1 * 1048576 = 1048576 bytes)

Console.WriteLine("Ingrese el tamaño del bloque fisico en KB: ");
int BloqueFisicoTamaño1 = int.Parse(Console.ReadLine()); //tamaño de 1
```

bloque en KB (ejemplo: 1KB)

```
BloqueFisicoTamaño1 *= 1024; //convierto de KB a bytes (ejemplo: 1 * 1024 = 1024 bytes)
```

```
int CantidadBloquesFisicos1 = DiscoRigidoTamaño1 / BloqueFisicoTamaño1;
Console.WriteLine($"Cantidad de bloques fisicos: {CantidadBloquesFisicos1}"); // (ejemplo: 1048576 / 1024 = 1024 Bloques fisicos)
```

```
Console.WriteLine("\nIngrese el nombre del archivo:");
string archnombre1 = Console.ReadLine();
```

```
Console.WriteLine("Ingrese el tamaño del archivo en KB:");
int archtamaño1 = int.Parse(Console.ReadLine()); // (ejemplo: 7 KB)
```

```
int BloqueEnlazadoComienzo = 10;
Console.WriteLine($"Bloque inicial del archivo: {BloqueEnlazadoComienzo}");
```

```
Console.WriteLine($"\\nIngrese el tamaño de los registros logicos en Bytes\\n(Preferiblemente menor a {BloqueFisicoTamaño1} Bytes): ");
int RegistroLogicoTamaño1 = int.Parse(Console.ReadLine()); //defino tamaño de registro logico (ejemplo: 100 bytes)
```

```
int FactorBloqueo1 = BloqueFisicoTamaño1 / RegistroLogicoTamaño1;
//Factor bloqueo
Console.WriteLine($"Factor bloqueo: {FactorBloqueo1}"); // (ejemplo = 10)
```

```
int archtamañobytes1 = archtamaño1 * 1024;
```

```
int CantBloqueArchivo1 = archtamañobytes1 / BloqueFisicoTamaño1; //al archivo divido por el tamaño de un bloque fisico (ejemplo: 7168 / 1024 = 7 con resto 0)
```

```
if ((archtamañobytes1 % BloqueFisicoTamaño1) > 0)
{
    CantBloqueArchivo1++; // si la division previa tiene resto mayor a 0, se necesita un bloque mas para contener el archivo (ejemplo: 7 + 0 = 7 bloques)
}
```

```
for (int i = 0; i < CantBloqueArchivo1; i++)
{
    BloqueEnlazadoComienzo += 2; //por cada bloque del archivo, sumo dos bloques para conseguir el numero par de bloques que pide el enunciado
}
```

```
Console.WriteLine($"\\n--Directorio--\\nNombre: {archnombre1} Bloque inicio: {10}, Cantidad de bloques {CantBloqueArchivo1}\\n");
```

```
Console.WriteLine("Ingrese el numero de registro logico relativo a 1: ");
int NumeroRegistroLogico1 = int.Parse(Console.ReadLine()); //elijo un
```

numero de reg logico (ejemplo = 45)

```
// El registro fisico 10 contiene los registros logicos de 1 a 10
// El registro fisico 12 contiene los registros logicos de 11 a 20
// El registro fisico 14 contiene los registros logicos de 21 a 30
// El registro fisico 16 contiene los registros logicos de 31 a 40
// El registro fisico 18 contiene los registros logicos de 41 a 50 (el registro
logico 45 va a estar en este registro fisico)
// El registro fisico 20 contiene los registros logicos de 51 a 60
// El registro fisico 22 contiene los registros logicos de 61 a 70

int RegistroFisicoQueContieneElRegistroLogico1 = (NumeroRegistroLogico1
/ FactorBloqueo1); //La parte entera de la division (ejemplo: 45/10 = 4 con resto 5)
int RegistroFisicoQueContiene1 =
(RegistroFisicoQueContieneElRegistroLogico1*2) + 10; //Le sumo la parte entera a
la direccion de comienzo (ejemplo: 10 + (4*2) = 18 )
Console.WriteLine($"El registro fisico que contiene al registro logico:
{RegistroFisicoQueContiene1}");

int PosRegLogDentroRegFis1 = NumeroRegistroLogico1 % FactorBloqueo1;
// Resto de la misma division de la linea 130 (ejemplo: 45/10 = 4 con resto 5)

int CantidadDeBloquesLeido =
RegistroFisicoQueContieneElRegistroLogico1; //uso la variable de la parte entera de
la division de la linea 130 para definir la cantidad de bloques que tendre que leer
if ((NumeroRegistroLogico1 % FactorBloqueo1) > 0)
{
    CantidadDeBloquesLeido++; // si la division previa tiene resto mayor a 0,
    se necesita leer el siguiente bloque para leer el registro logico en busqueda
    (ejemplo: 4 + 1 = 5 bloques)
}
Console.WriteLine($"\\nRegistros Fisicos leido hasta encontrar el registro
fisico que contiene el registro logico buscado: {CantidadDeBloquesLeido}"); //el
quinto bloque leido es el bloque 18

Console.WriteLine("Lista de numero de registros fisicos leidos para encontrar
el registro fisico que contiene el registro logico en busqueda: ");
for (int i = 0; i < CantidadDeBloquesLeido; i++)
{
    int bloque = 10 + (i * 2);
    Console.WriteLine($"Registro: {bloque}");
}

Console.WriteLine($"\\nPosicion del registro logico dentro del registro fisico
que lo contiene: {PosRegLogDentroRegFis1}"); //entonces el registro logico 45 esta
en la posicion 5 del registro fisico 18

Console.WriteLine("\\nPresione una tecla para Continuar a Asignacion
Indexada... ");
Console.ReadKey();
```

```
//----- INDEXADA -----  
  
//Asignacion Indexada  
Console.WriteLine("Asignacion Indexada\n\n");  
  
Console.WriteLine("Ingrese tamaño del Disco Rígido en MB: ");  
int DiscoRigidoTamaño2 = int.Parse(Console.ReadLine()); // ejemplo (1MB)  
  
DiscoRigidoTamaño2 *= 1048576; //convierbo de MB a bytes //ejemplo: 1 *  
1048576 = 1048576 bytes  
  
Console.WriteLine("Ingrese el tamaño del bloque físico en KB: ");  
int BloqueFisicoTamaño2 = int.Parse(Console.ReadLine()); //tamaño de 1  
bloque en KB (ejemplo: 1KB)  
  
BloqueFisicoTamaño2 *= 1024; //convierbo de KB a bytes (ejemplo: 1 * 1024  
= 1024 bytes)  
  
int CantidadBloquesFisicos2 = DiscoRigidoTamaño2 / BloqueFisicoTamaño2;  
Console.WriteLine($"Cantidad de bloques físicos:  
{CantidadBloquesFisicos2}"); //ejemplo: 1048576 / 1024 = 1024 Bloques físicos  
  
Console.WriteLine("\nIngrese el nombre del archivo: ");  
string archnombre2 = Console.ReadLine();  
  
Console.WriteLine("\nIngrese el tamaño del archivo en KB: ");  
int archtamaño2 = int.Parse(Console.ReadLine()); // (ejemplo: 7 KB)  
  
Console.WriteLine("\nIngrese el bloque de índice asignado al archivo:  
\n(Número par apartir del bloque 10)");  
int archBloqueDeIndices = int.Parse(Console.ReadLine()); // (ejemplo:  
Bloque 20)  
  
Console.WriteLine($"\\n--Directorio--\\nArchivo: {archnombre2}, Index Block:  
{archBloqueDeIndices}");  
  
Console.WriteLine($"\\nIngrese el tamaño de los registros lógicos en  
Bytes\\n(Preferiblemente menor a {BloqueFisicoTamaño2} Bytes): ");  
int RegistroLogicoTamaño2 = int.Parse(Console.ReadLine()); //defino tamaño  
de registro lógico (ejemplo: 100 bytes)  
  
int FactorBloqueo2 = BloqueFisicoTamaño2 / RegistroLogicoTamaño2;  
//Factor bloqueo  
Console.WriteLine($"\\nFactor bloqueo: {FactorBloqueo2}"); // (ejemplo = 10)  
  
int archtamañobytes2 = archtamaño2 * 1024;  
  
int CantBloqueArchivo2 = archtamañobytes2 / BloqueFisicoTamaño2; //al  
archivo divido por el tamaño de un bloque físico (ejemplo: 7168 / 1024 = 7 con resto
```

```
0)
    if ((archtamañobytes2 % BloqueFisicoTamaño2) > 0)
    {
        CantBloqueArchivo2++; // si la division previa tiene resto mayor a 0, se
        // necesita un bloque mas para contener el archivo (ejemplo: 7 + 0 = 7 bloques)
    }
    Console.WriteLine($"\\nEl bloque de indices permite almacenar
{FactorBloqueo2} indices que apuntan a bloques de datos del archivo.");

    Console.WriteLine($"\\nContenido del bloque de indice del archivo
{archnombre2}\\n");
    for (int i = 0; i < CantBloqueArchivo2; i++)
    {
        int bloque = 10 + (i * 2);
        if(bloque != archBloqueDeIndices)
        {
            Console.WriteLine($"El indice {i} del bloque de indice
{archBloqueDeIndices} apunta al registro fisico numero {bloque}");
        }
    }

    Console.WriteLine("\\nIngrese el numero de registro logico relativo a 1: ");
    int NumeroRegistroLogico2 = int.Parse(Console.ReadLine()); //elijo un
    numero de reg logico (ejemplo = 38 )

    //dentro del bloque de index numero 20

    // El indice 0 apunta al registro fisico 10 que contiene los registros logicos de
    1 a 10
    // El indice 1 apunta al registro fisico 12 que contiene los registros logicos de
    11 a 20
    // El indice 2 apunta al registro fisico 14 que contiene los registros logicos de
    21 a 30
    // El indice 3 apunta al registro fisico 16 que contiene los registros logicos de
    31 a 40 (el registro logico 38 va a estar en este registro fisico)
    // El indice 4 apunta al registro fisico 18 que contiene los registros logicos de
    41 a 50
    // El indice 5 apunta al registro fisico 22 que contiene los registros logicos de
    51 a 60
    // El indice 6 apunta al registro fisico 24 que contiene los registros logicos de
    61 a 70

    int RegistroFisicoQueContieneElRegistroLogico2 = (NumeroRegistroLogico2
    / FactorBloqueo2); //La parte entera de la division (ejemplo: 38/10 = 3 con resto 8 )
    int RegistroFisicoQueContiene2 =
    (RegistroFisicoQueContieneElRegistroLogico2 * 2) + 10; //Le sumo la parte entera a
    la direccion de comienzo (ejemplo: 10 + (3*2) = 16)
    Console.WriteLine($"El registro fisico que contiene al registro logico:
{RegistroFisicoQueContiene2}");
```

```
int PosRegLogDentroRegFis2 = NumeroRegistroLogico2 % FactorBloqueo2;  
// Resto de la misma division de la linea 130 (ejemplo: 38/10 = 3 con resto 8 )
```

Console.WriteLine(\$"\\nPosicion del registro logico dentro del registro fisico que lo contiene: {PosRegLogDentroRegFis2}"); //entonces el registro logico 38 esta en la posicion 8 del registro fisico 18

```
Console.WriteLine("\\nPrograma terminado... Presione una tecla para salir.");  
Console.ReadKey();
```

## Ejercicio 4

### Trabajo Practico N°3 - Arquitectura de Sistemas operativos

#### Punto 4)

Desde el punto de vista del rendimiento, en función del tiempo de acceso a disco para la recuperación de todo el archivo ¿Cuál considera como el más óptimo, explique los motivos de su conclusión?

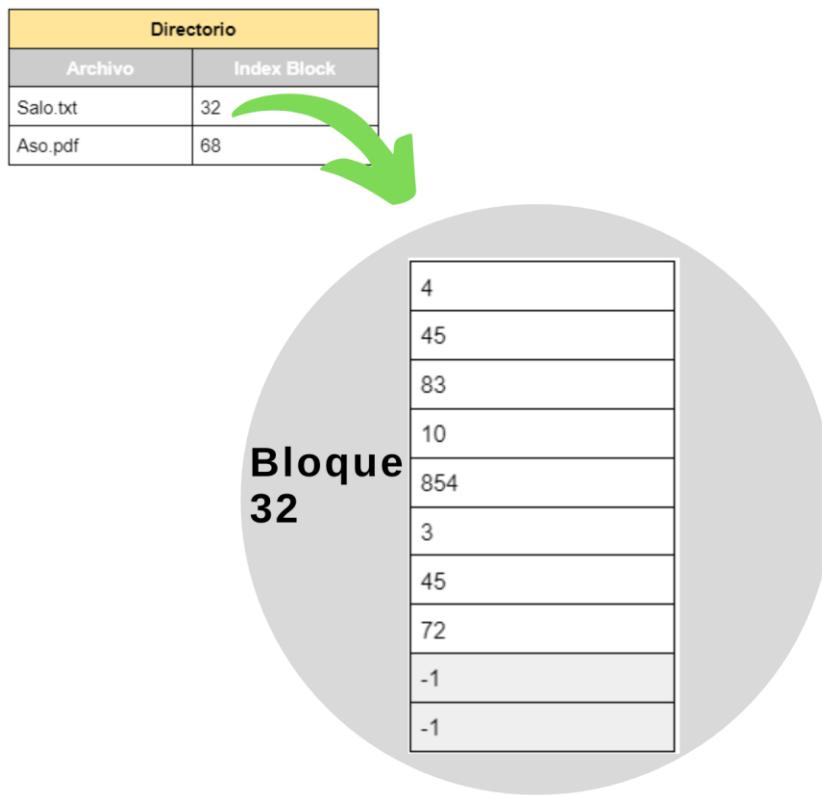
Desde el punto de vista del rendimiento, en función de acceso a disco para la recuperación de todo el archivo, el método de asignación más óptimo vendría a ser la **asignación indexada**.

Esto es porque para ingresar al contenido del archivo solamente necesita saber el index block de ese mismo archivo. El index block está en el directorio de archivos.

Directorio	
Archivo	Index Block
Salo.txt	32
Aso.pdf	68

Entonces el bloque 32 es un bloque de gestión que contiene punteros a los bloques de índice de datos del archivo Salo.txt.

Supongamos que el archivo Salo.txt es de 8 bloques de tamaño y el bloque de índice puede contener 10 índices a bloques de datos en total. El bloque de índice quedaría así:



Si el proceso requiere contenido del registro lógico que está contenido por el bloque físico número 45, solamente tendrá que ingresar al bloque de índice 32, luego al puntero 45 para entrar directo al registro 45 donde contendrá el registro lógico del archivo Sal0.txt en búsqueda.

Con respecto a la recuperación de todo el archivo, la indexada es la mas rápida por que solamente ingresando el bloque de índices, donde contiene punteros a todos los registros físicos que contiene datos del archivo en búsqueda, vendría a ser la más óptima en función de tiempo y acceso

En cambio, en **asignación enlazada**, el directorio muestra el archivo con su bloque de comienzo y bloque de fin.

Esto quiere decir que si estamos buscando datos del registro lógico que está dentro de un bloque físico, y este mismo bloque está a lo último de la cola de registros físicos del archivo, **el sistema tendrá que leer todos esos registros físicos a partir del bloque de comienzo hasta llegar al registro físico que contiene el registro lógico en búsqueda.**

La **asignación contigua** tiene su ventaja y su desventaja.

La **ventaja** que tiene es el acceso directo. Se puede recorrer un archivo muy rápidamente en forma secuencial y también puedes encontrar muy rápidamente un registro del archivo en forma directa.

El **problema** con almacenamiento contigua esto es que si el archivo **Sal0.txt** quiere crecer y necesita 5 bloques más, no tendrá ese espacio solicitado porque estaría chocando con otro archivo.

Directorio (Contigua)		
Archivo	Start	Length
Sal0.txt	32	32
Aso.pdf	68	104

Si el archivo Sal0.txt crece y solicita 6 bloques más, estaría chocando con el archivo Aso.pdf

Comienzo (32) + Length (32) = 64 bloques contiguas asignado al archivo Sal0.txt.

El archivo crece y necesita usar 6 bloques más →  $64 + 6 = 70$

Pero el archivo Aso.pdf empieza en el bloque 68 entonces estarían chocando.

En mi opinión el más óptimo en general vendría a ser el almacenamiento indexado por lo dinámico que es en la búsqueda de datos teniendo en cuenta que no cuenta con fragmentaciones externas o internas.