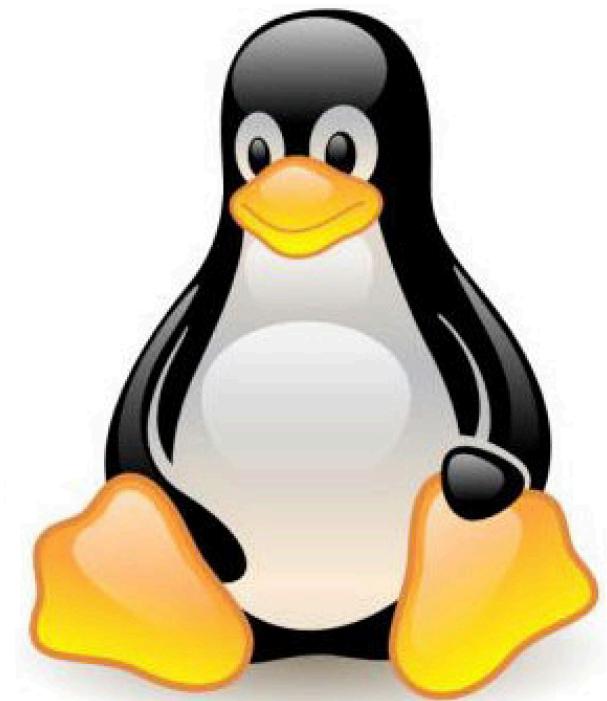
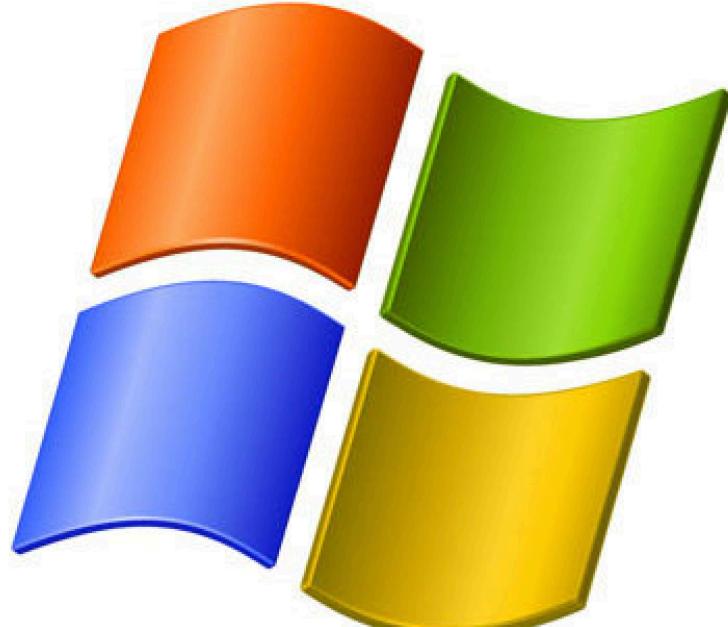


Tema 2. Primera parte: Procesos

Sistemas Operativos

ETSI Informática. Universidad de Málaga



Manuel Ujaldón

Catedrático de Arquitectura de Computadores
Departamento de Arquitectura de Computadores
Universidad de Málaga

Índice [41 diapositivas]

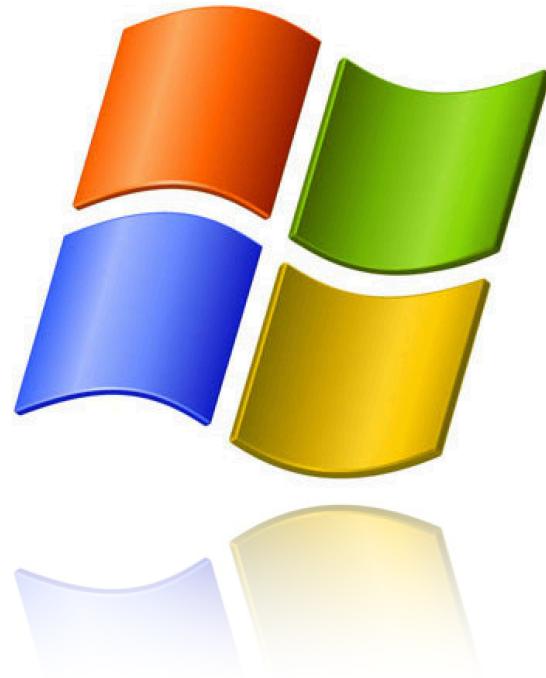
I. Procesos [23]

1. Concepto
2. Estados
3. Bloque de control del proceso
4. Creación y terminación de procesos

II. Hilos (threads) [13]

1. Concepto
2. Librerías para crear procesos

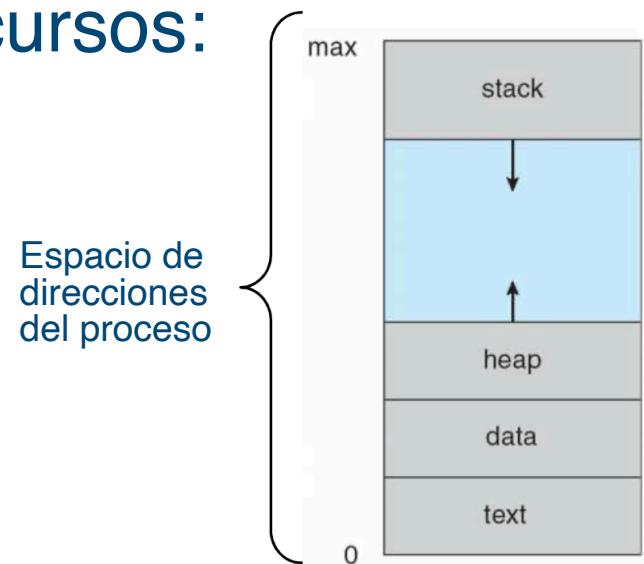
III. Ejemplos en los S.O actuales [5]



I. Procesos

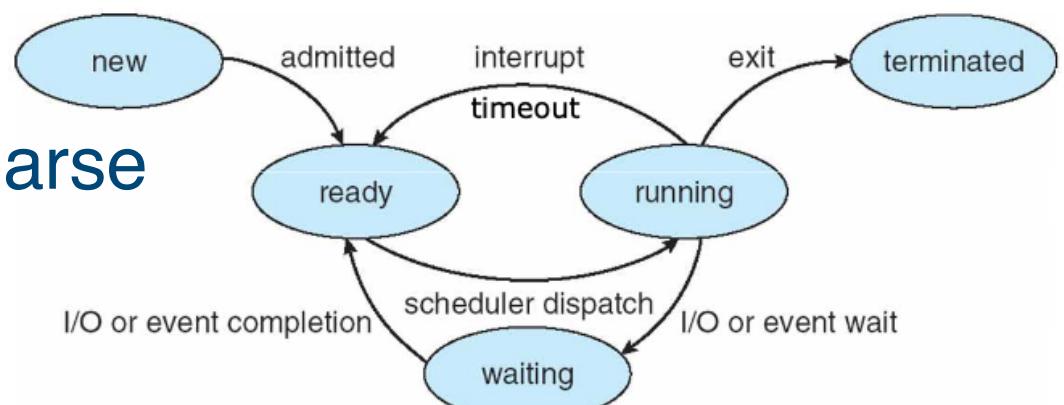
Concepto de proceso

- El S.O. ejecuta una gran variedad de tareas, que describen mediante programas cómo llevarlas a cabo.
- El proceso es el programa en ejecución, y por tanto, de un mismo programa pueden surgir varios procesos como instancias vivas de él.
- Cada proceso se identifica mediante un número, su PID.
- Un proceso necesita los siguientes recursos:
 - Memoria para alojar el programa.
 - Registros de la CPU para valores de la ejecución (PC, AX, BX, ...).
 - La pila, para datos temporales, paso de parámetros a funciones y direcciones de retorno de ellas.



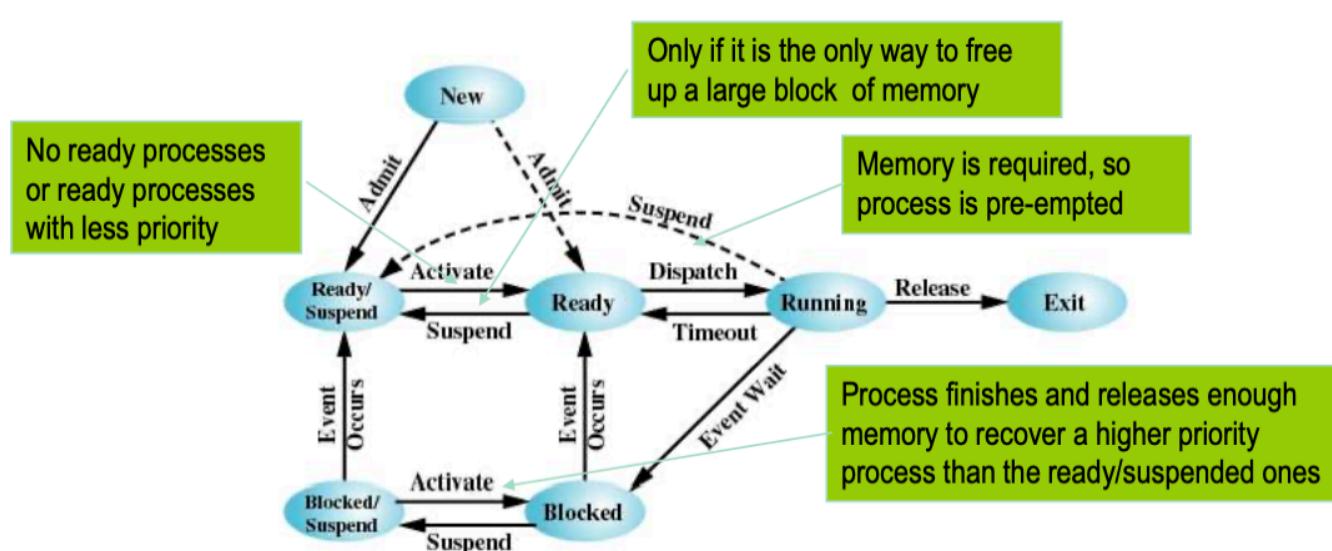
Estados de un proceso

- Un proceso va pasando por diferentes estados durante su ejecución, que el S.O. debe encargarse de mantener.
- Un modelo basado en 5 estados es el siguiente:
 - **New:** El proceso está siendo creado.
 - **Running:** Se están ejecutando sus instrucciones.
 - **Waiting:** El proceso está bloqueado esperando algún evento
 - **Ready:** El proceso está esperando a que se le asigne una CPU.
 - **Terminado:** Su ejecución ha finalizado.
- La evolución del proceso por sus estados puede reflejarse en el siguiente diagrama:



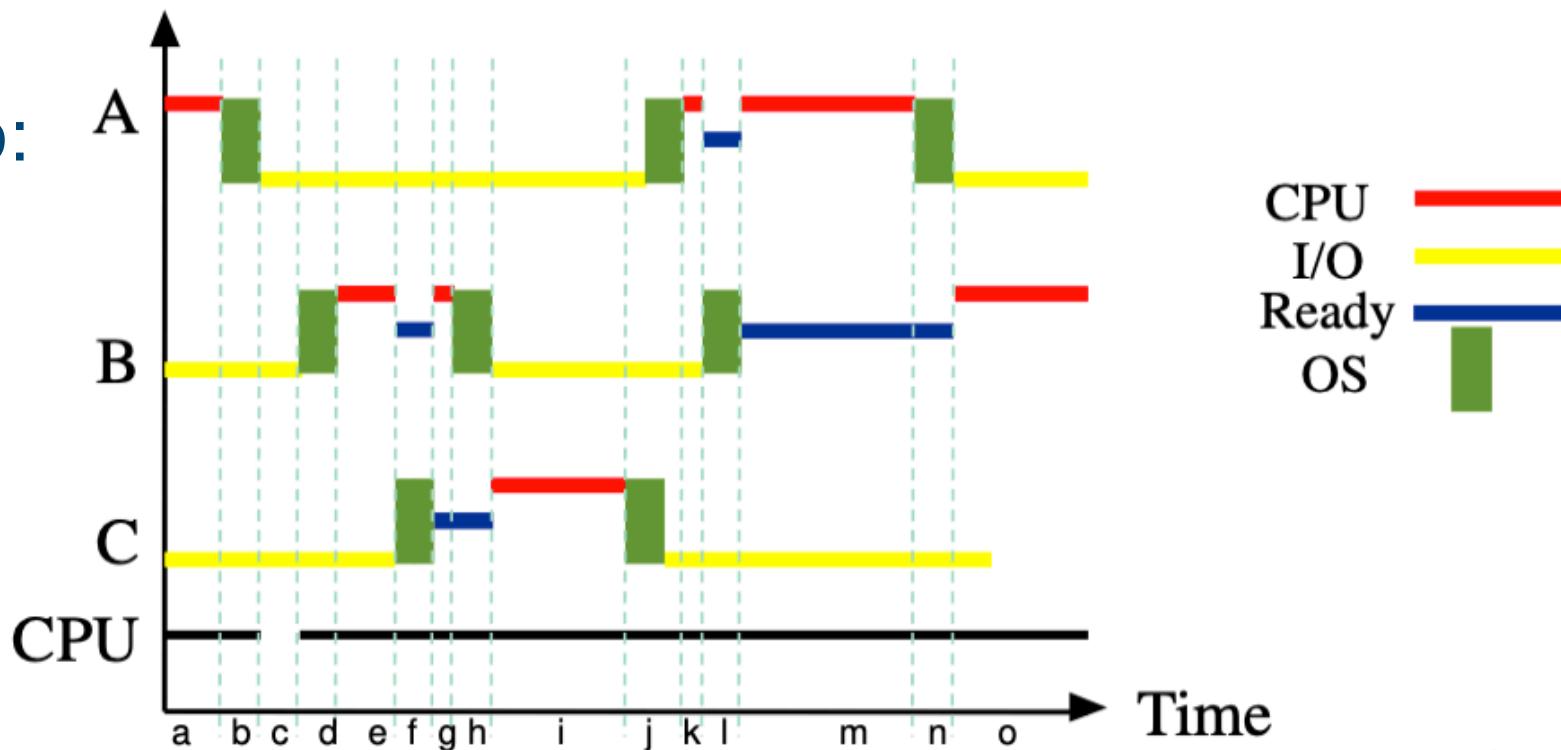
Suspensión de procesos

- En los sistemas de memoria virtual, los procesos pueden trasladarse temporalmente a disco para liberar su memoria, lo que debe reflejarse contemplando dos nuevos estados:
 - Blocked/Suspend: El proceso se ha trasladado a disco mientras estaba bloqueado en espera de algún evento.
 - Ready/Suspend: El proceso se ha trasladado a disco cuando estaba listo para volver a usar la CPU (esperando su turno).



Seguimiento de los procesos

Ejemplo:



a: A en CPU, B y C bloqueados.

b: A solicita al S.O. una operación de E/S.

c: Todos los procesos bloqueados (**CPU ociosa**)

d: B finaliza su E/S (se desbloquea).

e: B en ejecución.

f: C finaliza su E/S (se desbloquea), B está listo.

g: B continúa en la CPU, C está listo.

h: B system call y se bloquea. C ocupa la CPU.

i: C tiene la CPU, A y B están bloqueados.

j: C solicita E/S al S.O. A se desbloquea.

k: A en ejecución.

l: Una int. de E/S llama al S.O. para desbloquear B.

m: A continúa en ejecución y B en espera.

n: A se bloquea.

o: B comienza su ejecución.



El bloque de control del proceso (PCB: Process Control Block)

- Cuando la CPU conmuta del proceso A al B necesita alojar cierta información de A para retomarlo en un futuro.
- Esta información debe ser mantenida por el S.O. para caracterizarlo a lo largo de su ejecución, y constituye el PCB del proceso, que consta de los siguientes campos:
 - Estado en el que se encuentra.
 - Valores de los registros de la CPU (PC, SP, ...).
 - Info sobre la planificación del proceso (prioridades, colas, ...).
 - Uso de la memoria.
 - Información contable: Uso de la CPU, edad, limitaciones temporales.
 - Información del estado de su E/S: Dispositivos que tiene asignados, ficheros que tiene abiertos, ...

Registro de los PCBs por parte del S.O.

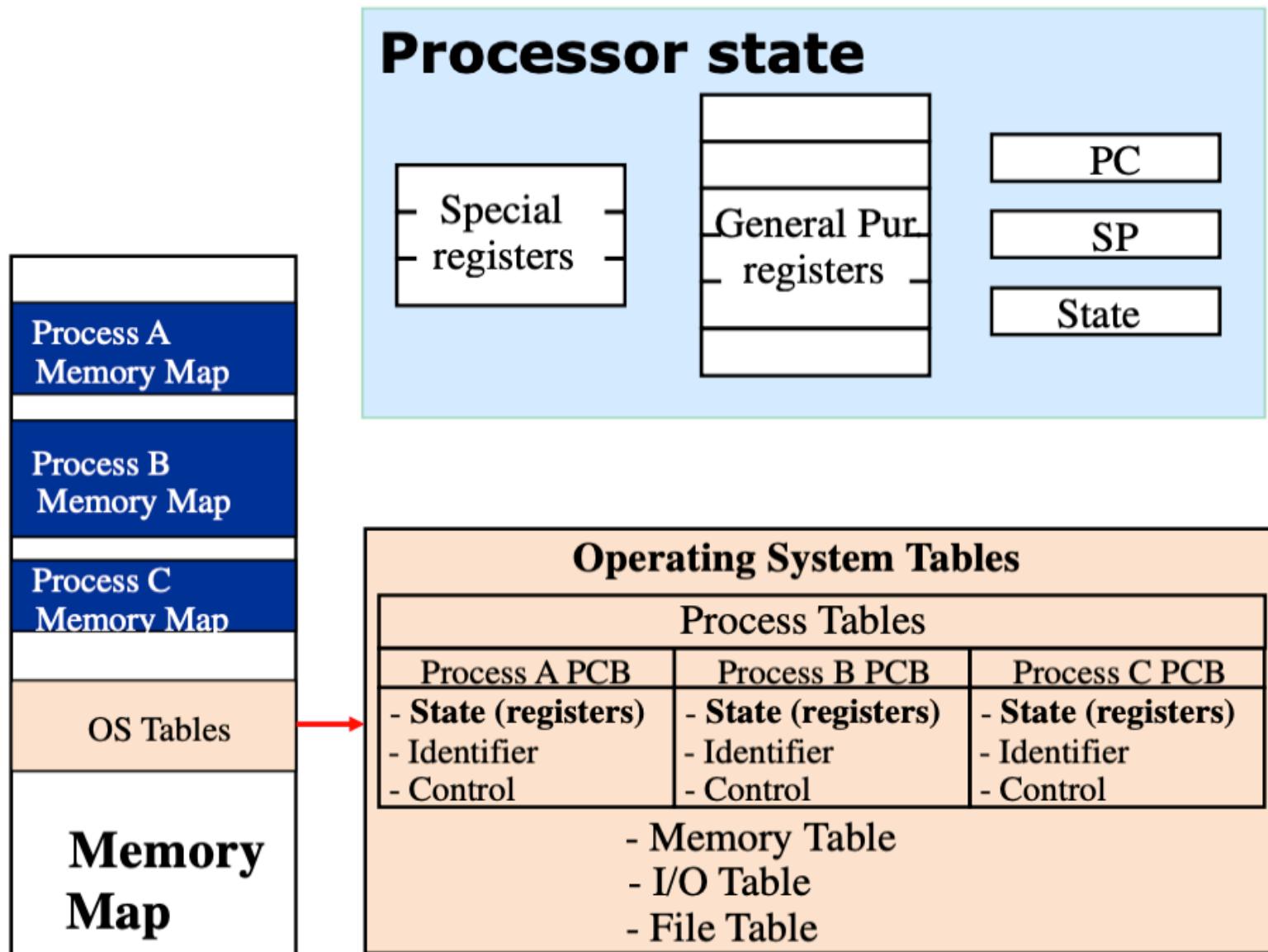
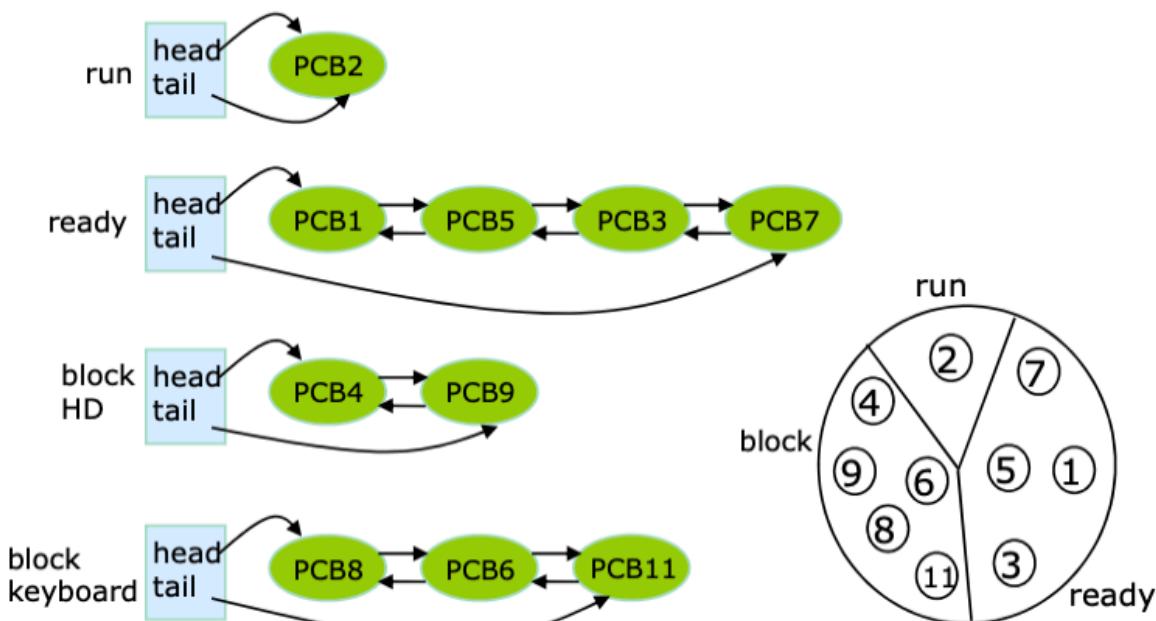


Imagen de un proceso

- Delimita el conjunto de información que lo caracteriza.
- Se compone de tres elementos principales:
 - El estado de la CPU, que refleja cómo estaba justo antes de interrumpir su ejecución.
 - Los datos de sus segmentos de código, datos y pila.
 - El bloque de control del proceso.

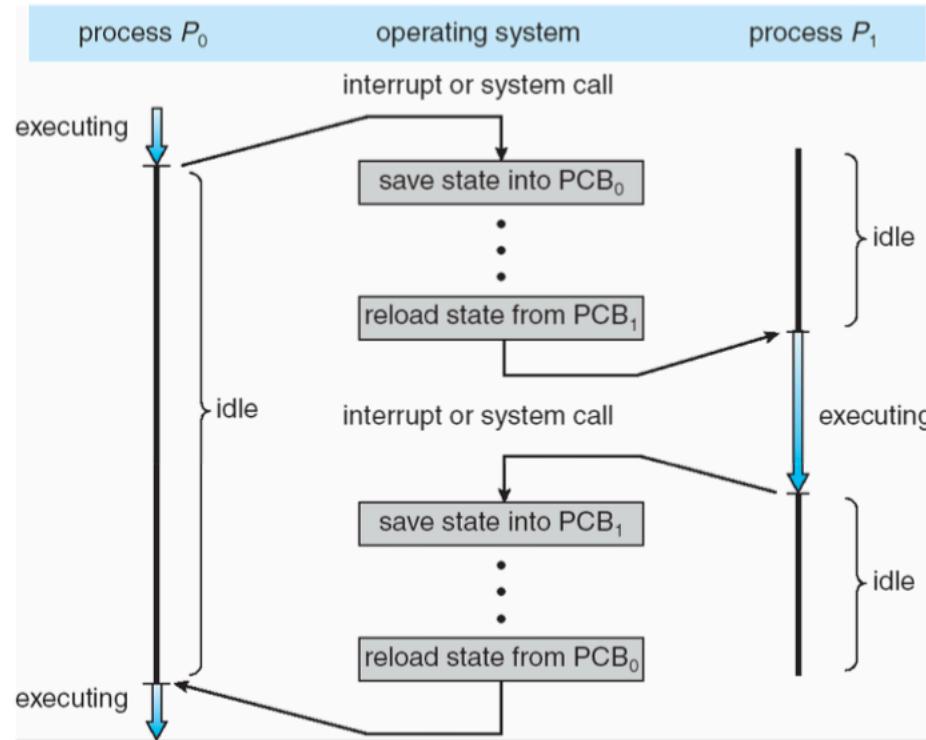
Organización de los procesos por parte del S.O.

- El S.O. mantiene una serie de listas donde ubica cada proceso atendiendo a su estado, y que son las siguientes:
 - Una lista “Run” por cada CPU disponible.
 - Una lista “Ready” que ordena el planificador de procesos.
 - Varias listas de procesos bloqueados atendiendo a ciertos criterios para acelerar la selección del proceso a desbloquear en cada caso.



Commutación de la CPU de un proceso a otro

- Se graba el PCB del proceso a desalojar y se carga el PCB del proceso entrante.
- Es un tiempo baldío al que el S.O. puede dedicar hasta un milisegundo, siendo mayor cuanto más complejo sea el PCB.

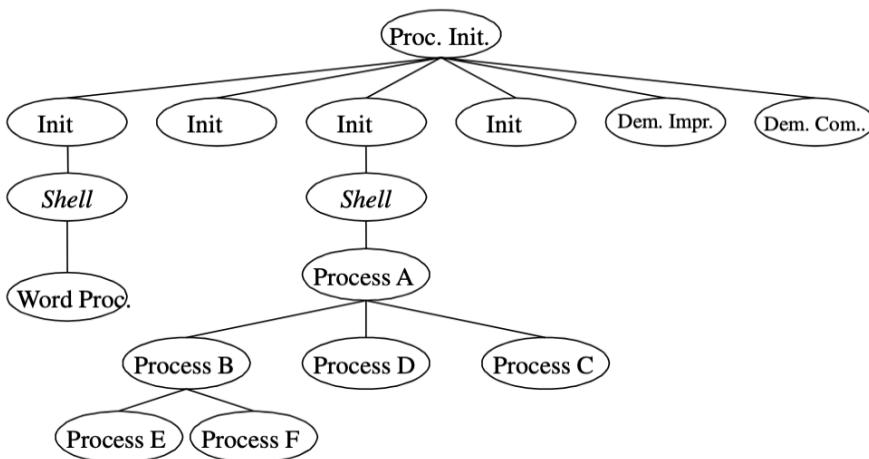


Creación de procesos

- Principales eventos que provocan la creación de procesos:
 - Inicialización del sistema.
 - Inicialización de un trabajo por lotes.
 - Petición de un usuario para crear un nuevo proceso.
 - Ejecución de la llamada al sistema para la creación de un proceso desde otro proceso.
- En UNIX (no así en Windows), un proceso padre puede crear procesos hijo, que a su vez pueden crear nietos, conformando un árbol o jerarquía de procesos.
 - Cada proceso tendrá su PID propio.
 - Llamadas al sistema permiten a los procesos comunicarse entre sí.
 - Padre e hijo se ejecutan concurrentemente.
 - El padre puede bloquearse en espera de que finalice el hijo.

Compartición de recursos entre procesos

- UNIX: El hijo clona el espacio de direcciones del padre, sustituyendo luego el programa del padre por el suyo. Esto facilita mucho la sincronización entre ellos.
 - La jerarquía que nace de un proceso se denomina “process group”.
 - El proceso inicial se denomina “Init”, que crea *daemons*, entre ellos el *shell*, desde el que el usuario lanza nuevos procesos.



- Windows: Todos los procesos se crean de la misma forma.
 - No se establece jerarquía alguna.

Creación de procesos en UNIX mediante el API POSIX

- Pueden utilizarse dos llamadas básicas.

- `pid_t fork(void)` crea un nuevo proceso que alberga una copia del espacio de direcciones del padre. A partir de ahí, la ejecución se desdobra en dos procesos, devolviendo:

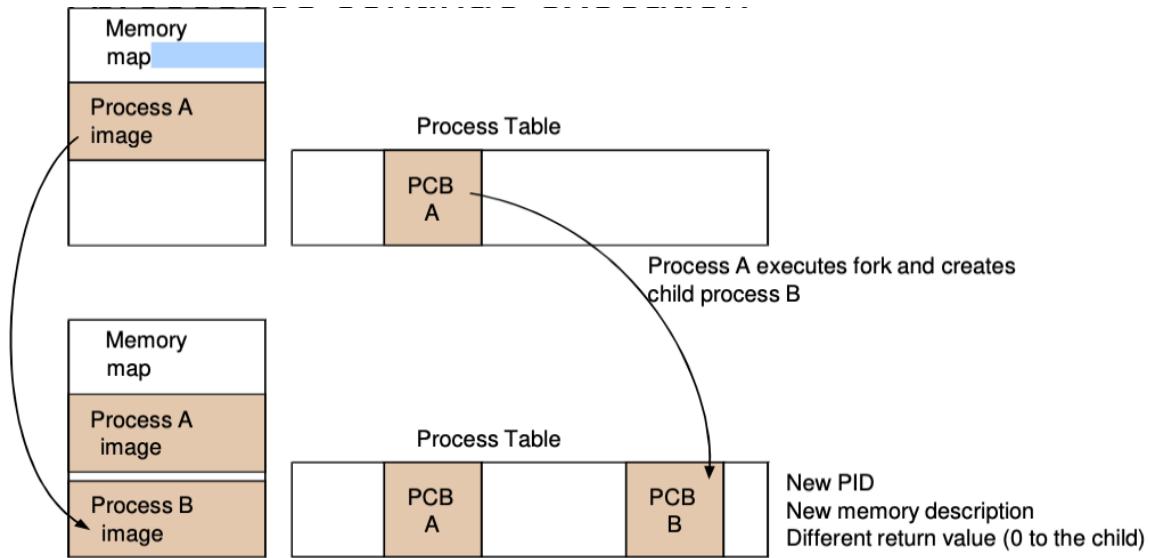
- El valor 0 en el proceso por donde prosigue el hijo.
 - El PID del hijo en el proceso por donde prosigue el padre.
 - Un valor -1 en caso de que se produzca un error.

- `exec()` permite reemplazar el programa del padre por el del hijo, que también puede proporcionarse mediante un comando o fichero (`execl/execlp`) junto a sus argumentos, que a su vez pueden proporcionarse de forma directa o indirecta (`execv/execvp`):

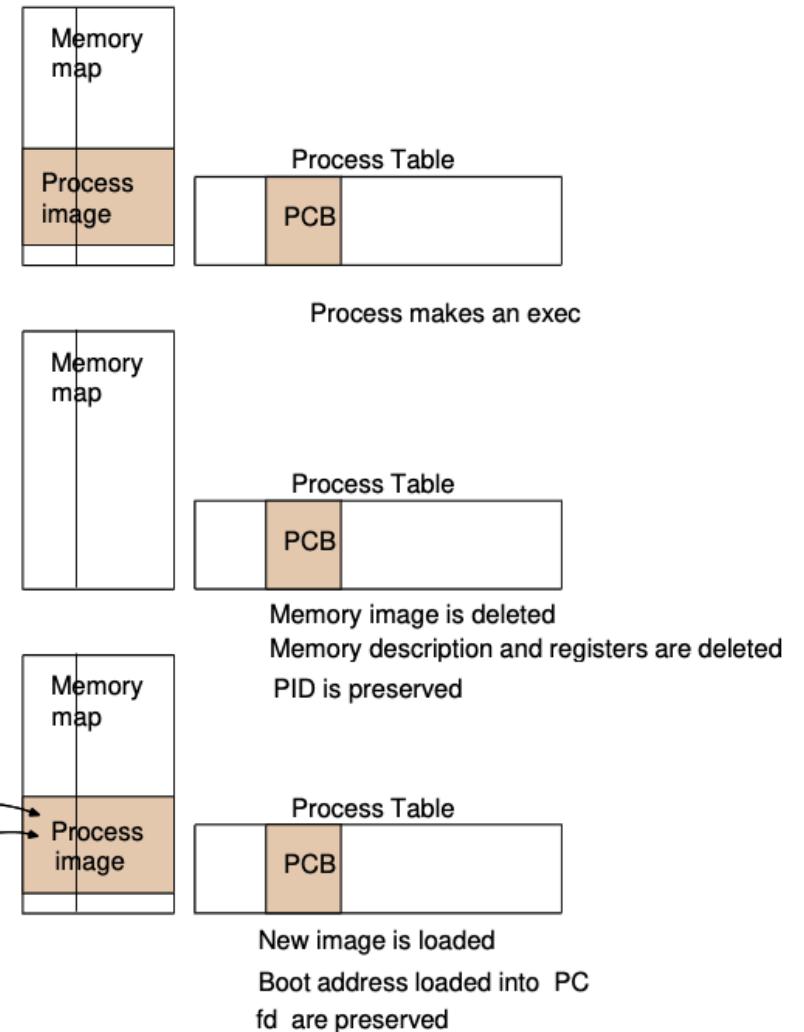
- `int execl(const char *path, const char *arg, ...);`
 - `int execlp(const char *file, const char *arg, ...);`
 - `int execv(const char *path, char *const arg, ...);`
 - `int execvp(const char *file, char *const arg, ...);`

Gestión del S.O. para fork() y exec()

fork()



exec()



Otras llamadas útiles

● Espera del padre a que finalice un hijo:

- `pid_t wait(int *status)` - Devuelve el PID del hijo que acaba (si el padre ha creado varios hijos, puede saber de cuál se trata)

● Llamadas al sistema para obtener PIDs (en el API POSIX):

- `pid_t getpid()` - Devuelve el PID del proceso.
- `pid_t getppid()` - Devuelve el PID del proceso padre.

● Finalización de procesos (en UNIX):

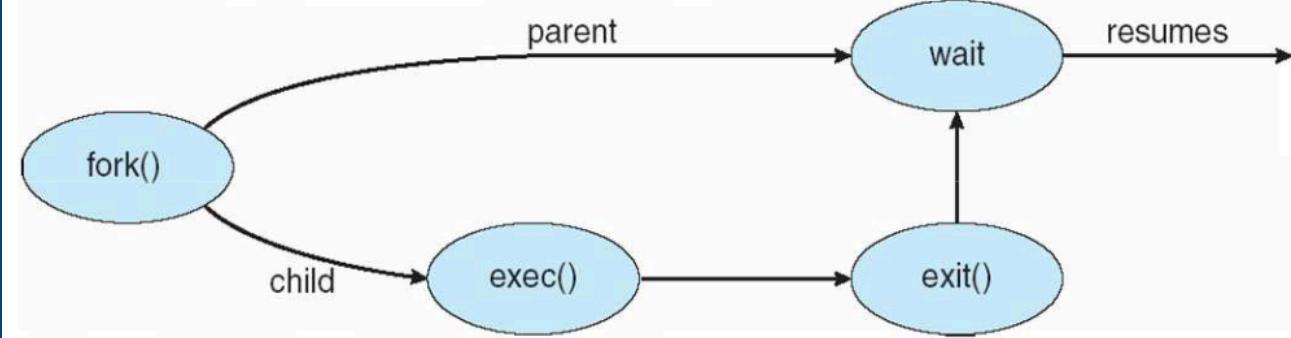
- `exit(int status)` - Termina y solicita al S.O. que libere sus recursos (el argumento es el código que se devuelve al padre en `wait`).
- `abort()` - Termina la ejecución de un proceso hijo de forma abrupta (por ejemplo, por haberse excedido en el uso de los recursos).
- Si el padre termina, en algunos S.O. se produce una finalización en cascada de todos sus hijos.

Ejemplo de uso

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

int main()
{
    pid_t pid;

    pid = fork();                      /* fork a child process */
    if (pid < 0) {                     /* error occurred */
        fprintf(stderr, "Fork failed");
        return 1;
    }
    else if (pid == 0) {               /* child process */
        execp("/bin/ls", "ls", "-l", NULL);
    }
    else {                            /* parent process */
        wait();                        /* parent will wait for the child to complete */
        printf("Child finished");
    }
    exit(0);
}
```



```

graph LR
    A(fork()) -- parent --> B(wait())
    A -- child --> C(exec())
    C --> D(exit())
    D -- resumes --> B
  
```

Mejora del ejemplo anterior. El padre espera a la finalización de ese hijo en exclusiva

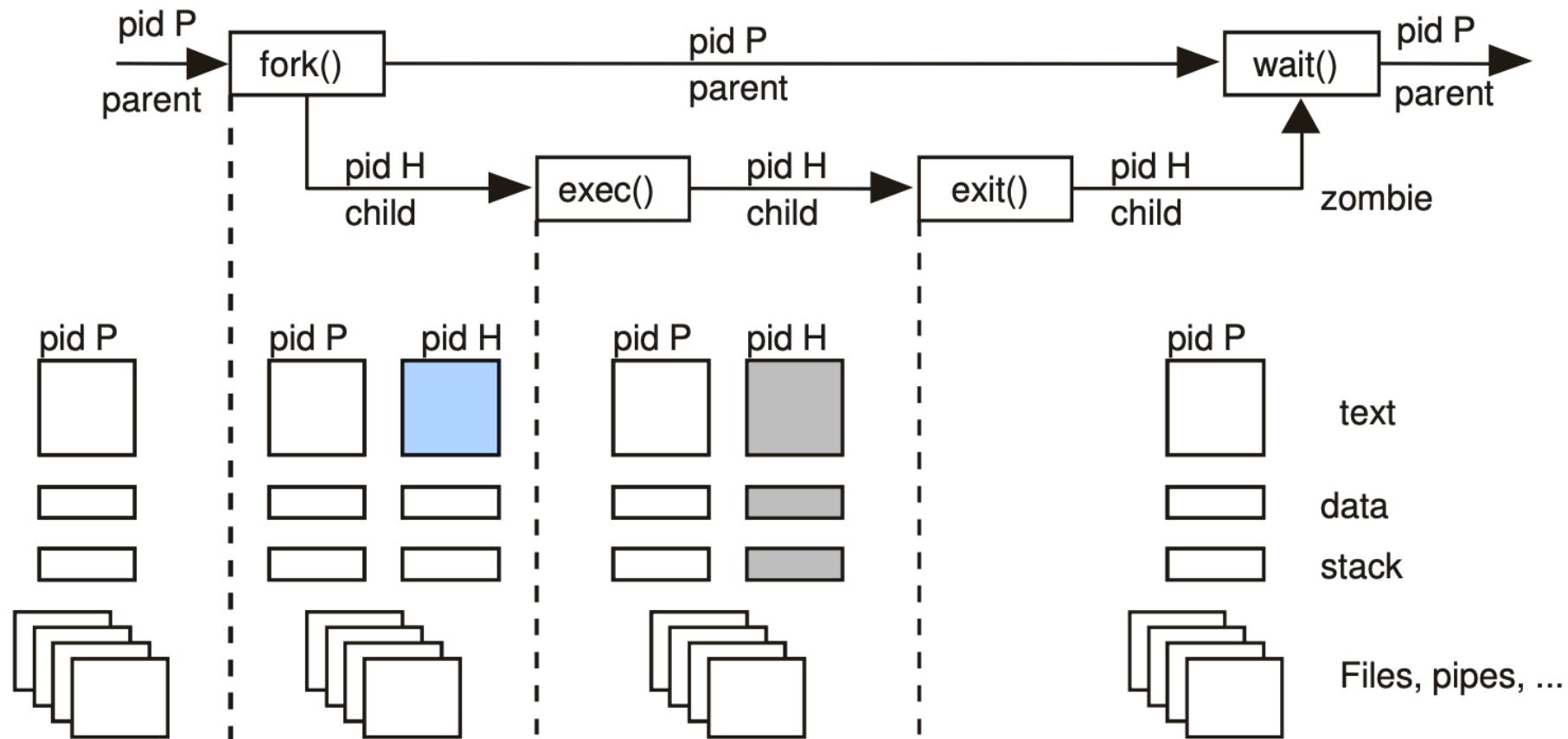
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int status;

    pid = fork();                  /* fork a child process */
    if (pid == 0) {                /* child process */
        execlp("/bin/ls", "ls", "-l", NULL);
        exit(-1);
    }
    else {                        /* parent process: will wait only for */
        while (pid != wait(&status)); /* that particular child to end */
        printf("Child finished");
    }
    exit(0);
}
```

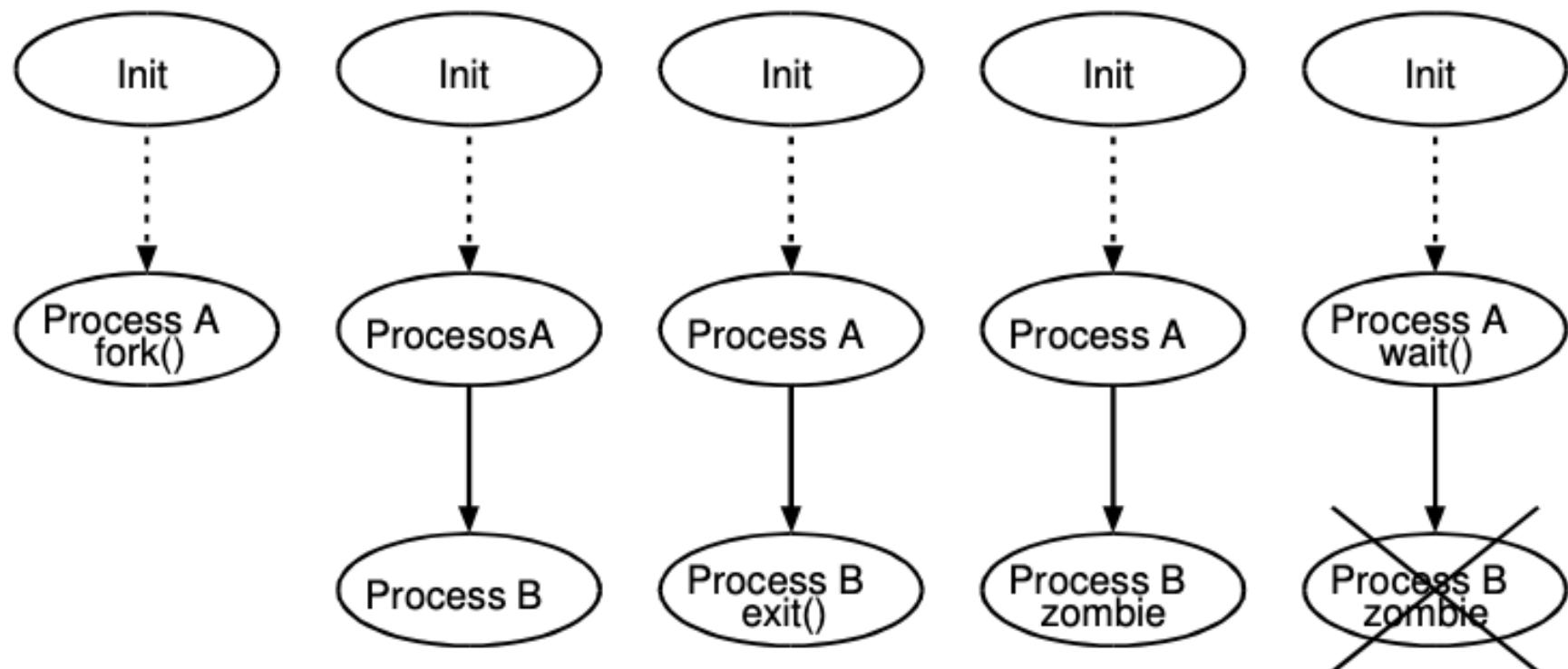
Evolución de los procesos padre e hijo

- El hijo se clona del padre y luego actualiza su programa.
- El padre puede esperar bloqueado a la finalización del hijo.



Finalización de los procesos en UNIX

- Cuando un proceso finaliza, a todos sus hijos se les asigna el proceso init como padre.
- La finalización de un hijo sin que el padre llame a wait() se considera una anomalía tipificada como proceso *zombie*.



Concepto de señal

- Es un mecanismo empleado en el S.O. para notificar a un proceso que ha ocurrido un determinado evento. La señal puede provenir del propio S.O. o de otro proceso que ejecute la llamada al sistema `kill()`.

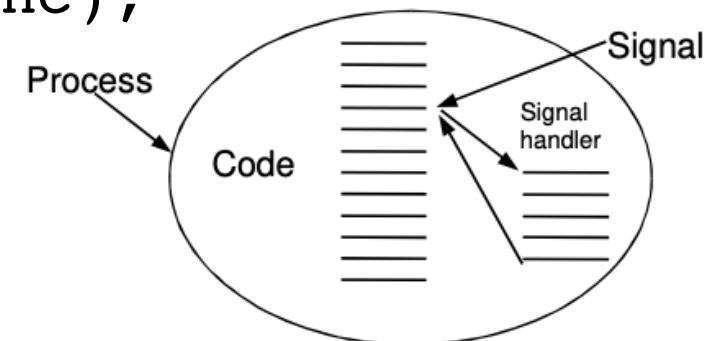
● `int kill(pid_t pid, int sign)`

- La llamada al sistema `signal()` indica la función a ejecutar cuando el proceso recibe dicha señal.

● `sig_t signal(int sign, sig_t func);`

- `signal()` es una versión reducida de `sigaction()`, pero suficiente en la mayoría de casos prácticos.

- `pause()` bloquea un proceso hasta que reciba una señal



Tipos de señales

Apuntamos las 10 más útiles, figurando en rojo las que usaremos en nuestro Shell de prácticas

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
4	SIGILL	create core image	illegal instruction
5	SIGTRAP	create core image	trace trap
6	SIGABRT	create core image	abort program (formerly SIGIOT)
7	SIGEMT	create core image	emulate instruction executed
8	SIGFPE	create core image	floating-point exception
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
15	SIGTERM	terminate process	software termination signal
16	SIGURG	discard signal	urgent condition present on socket
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
21	SIGTTIN	stop process	background read attempted from control terminal
22	SIGTOU	stop process	background write attempted to control terminal
23	SIGIO	discard signal	I/O is possible on a descriptor (see fcntl(2))
24	SIGXCPU	terminate process	cpu time limit exceeded (see setrlimit(2))
25	SIGXFSZ	terminate process	file size limit exceeded (see setrlimit(2))
26	SIGVTALRM	terminate process	virtual time alarm (see setitimer(2))
27	SIGPROF	terminate process	profiling timer alarm (see setitimer(2))
28	SIGWINCH	discard signal	Window size change
29	SIGINFO	discard signal	status request from keyboard
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

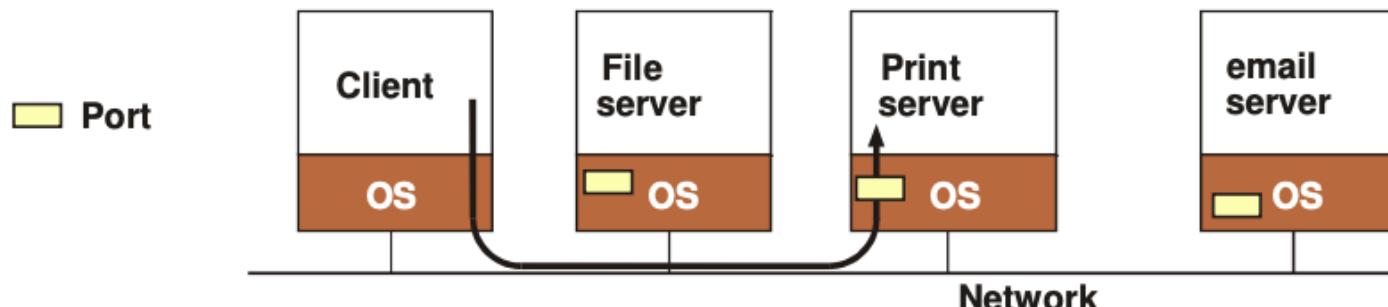
Procesos *daemon* (o agentes del sistema)

• Son procesos especiales:

- Se ejecutan en segundo plano.
- No están asociados a un terminal o proceso de entrada.
- Se quedan esperando un evento (la solicitud del cliente).
- Realizan una operación específica en momentos predeterminados.

• Características:

- Comienzan al iniciar el sistema y nunca mueren.
- No realizan la tarea en sí, sino que crean el proceso que la acomete.
- Pueden ubicarse en una máquina diferente a la del cliente.



Procesos cooperativos

- Son procesos que pueden afectar o ser afectados por la ejecución de otros procesos.
- Ventajas de la cooperación entre procesos:
 - Compartición de información.
 - Aceleración de la computación a través de subtareas paralelas.
 - Modularidad dividiendo las funciones del sistema en procesos aparte.
 - Comodidad. Incluso un proceso individual puede querer editar, imprimir y compilar en paralelo.
- Los procesos cooperativos necesitan articular mecanismos de comunicación interproceso (IPC) para intercambiar datos.
- Hay dos modelos básicos de IPC:
 - Memoria compartida.
 - Pase de mensajes.

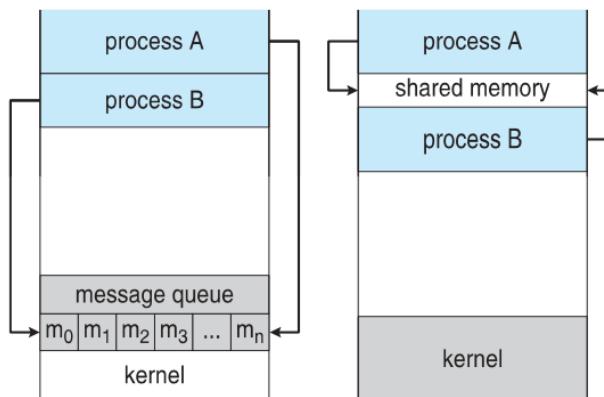
Comparativa entre pase de mensajes y memoria compartida

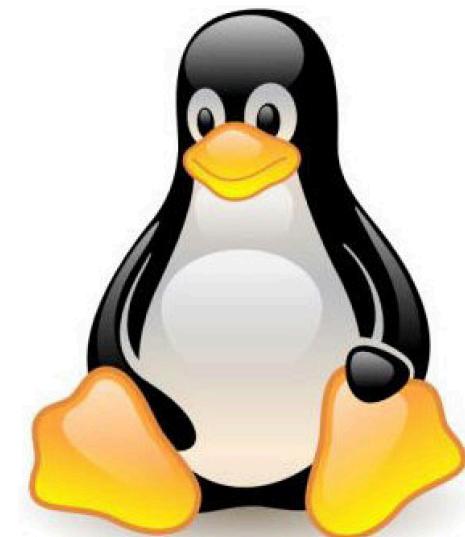
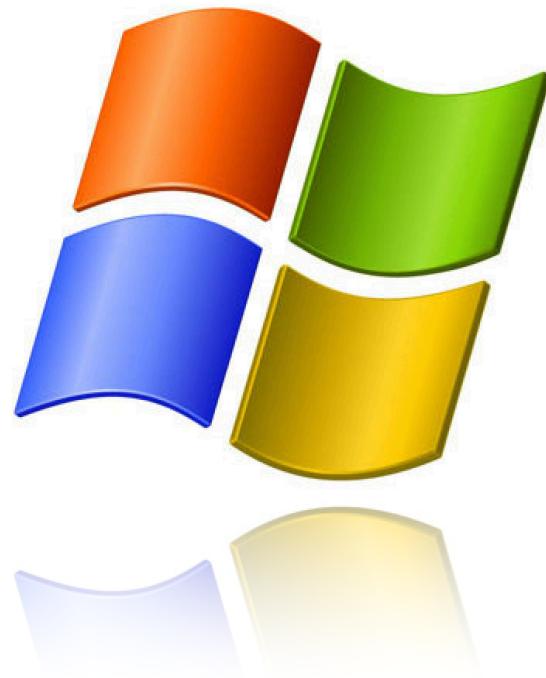
● Pase de mensajes:

- Muy útil para pequeñas cantidades de datos.
- Más sencillo de implementar que la memoria compartida.
- Requiere llamadas al sistema, y por tanto, la intervención del kernel.

● Memoria compartida:

- Mayor velocidad de la memoria y comodidad.
- Las llamadas al sistema sólo se requieren para establecer las regiones de memoria compartida. A partir de ahí, la E/S va por libre.

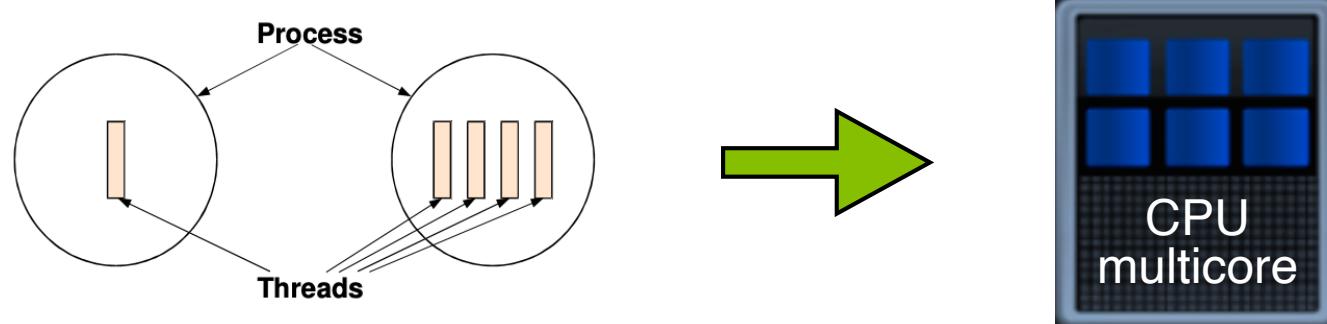




II. Hilos

Concepto de hilo (*thread*)

- Los procesos comparten la CPU, pero ¿qué ocurre cuando aparece la CPU multi-core? Para que un proceso pesado pueda usar varios cores, debe poder ramificarse en **hilos** que compartan el espacio de direcciones, facilitando así su creación, comunicación y compartición de recursos.
- El proceso retiene la propiedad de los recursos (memoria, ficheros, periféricos), mientras que sus hilos despliegan el paralelismo entre los cores para acelerar el proceso, y son los elegidos por el planificador de cada uno de estos cores.

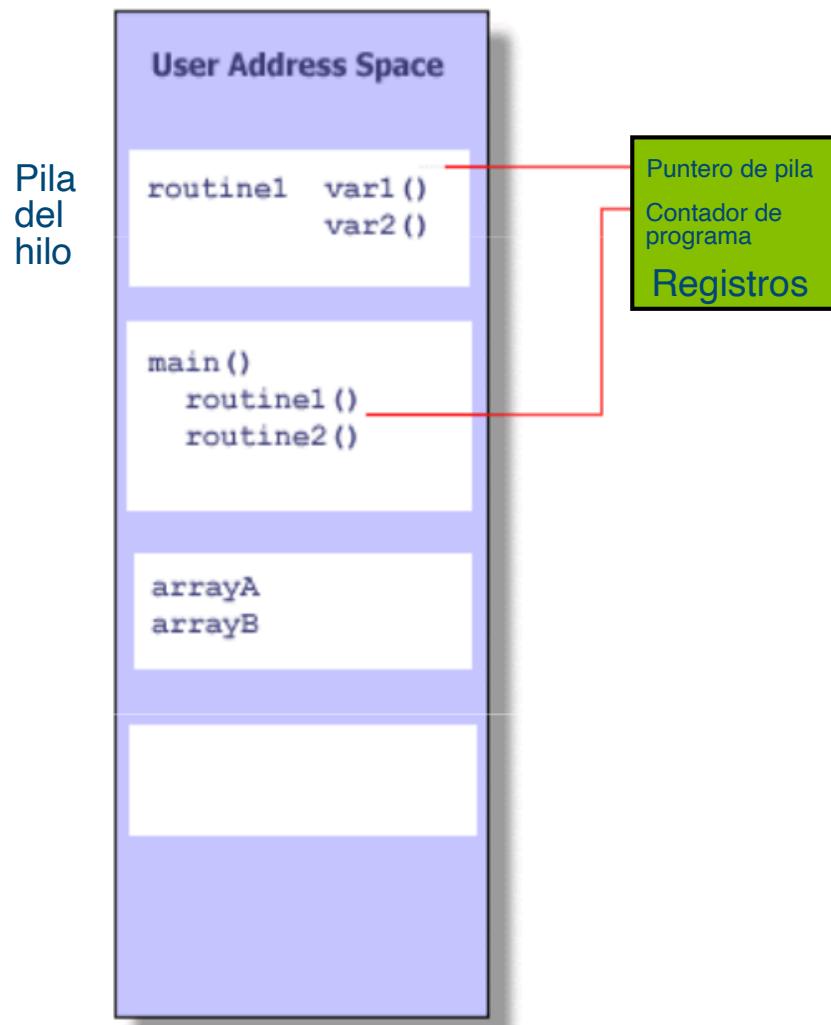


Definición de hilo e información asociada

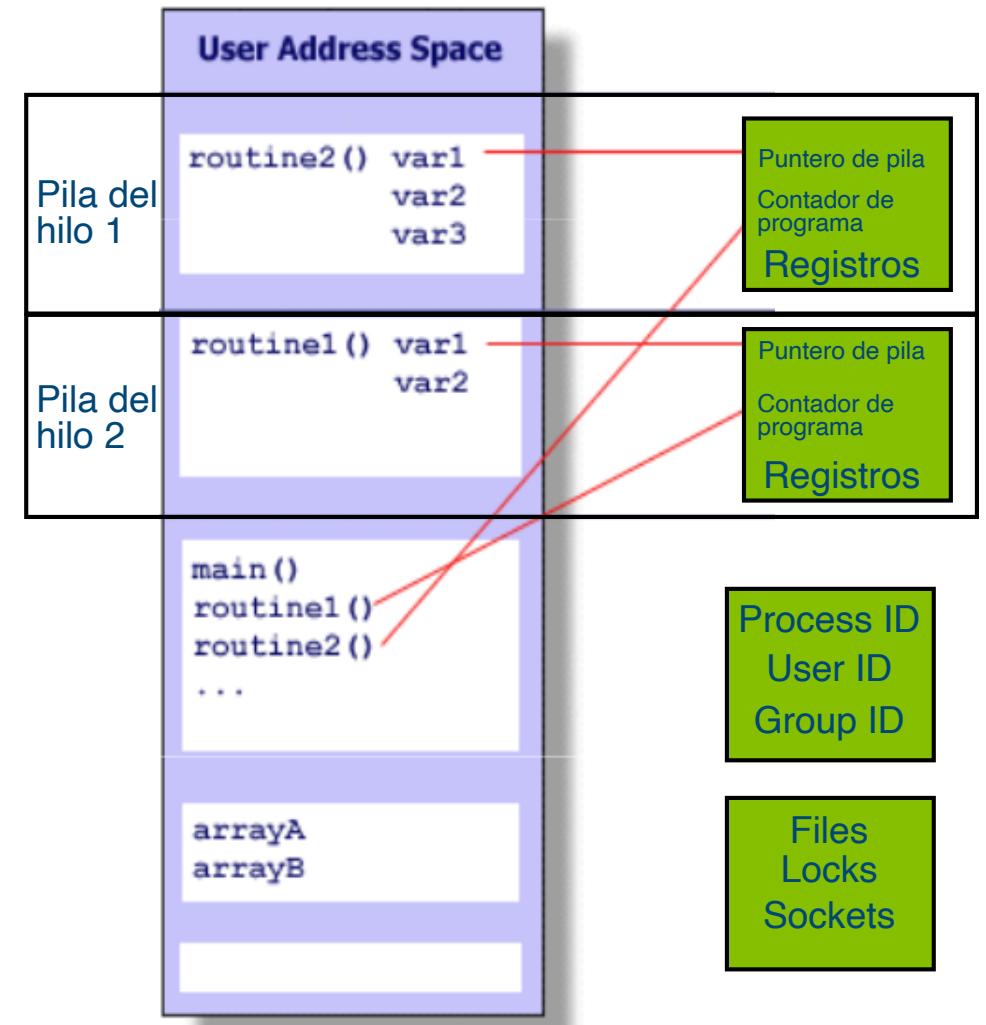
- El hilo es la entidad de un proceso que se planifica para ejecución. El proceso nace como un solo hilo, y a partir de ahí puede ramificarse en multitud de ellos de forma sucesiva.
- El S.O. mantiene la siguiente información para cada hilo:
 - Su ID, conjunto de registros y pila.
 - Su estado de ejecución (Running, Ready, ...) y almacenamiento estático para sus variables locales.
 - Su contexto guardado en memoria cuando no está en ejecución.
 - Cada hilo comparte con sus hermanos:
 - Los tres segmentos del espacio de direcciones del proceso (código, datos y pila).
 - Los recursos que tiene asignados el proceso (ficheros, dispositivos, ...).
 - Los procesos hijo, las variables globales y las señales, entre otras cosas.

Espacio de direcciones para los hilos

Proceso con un solo hilo



Proceso con dos hilos



Uso de los hilos

- Todas las aplicaciones se programan hoy en día multihilo.
Un par de ejemplos:

- Procesadores de texto: Uno lleva la corrección gramatical, otro muestra los gráficos, otro lee las pulsaciones de teclado, ...

- Navegadores Web: Uno visualiza las imágenes, otro recibe los datos por la red, otro atiende el interfaz de usuario, ...

- ¿Cómo implementaríamos las múltiples pestañas de un navegador Web?

- Si lo hacemos con hilos, comparten la memoria con el riesgo de que algunos puedan modificar una variable local y afectar a otros.

- Si lo hacemos con procesos, no comparten la memoria, ganando en fiabilidad. Tanto Chrome como Firefox optan por esta opción.

Multithreading

- Es la habilidad de un S.O. para soportar múltiples hilos concurrentes dentro de un mismo proceso.

● Beneficios:

- Interactividad: Puede proseguir la ejecución aunque parte de un proceso esté bloqueado.
- Compartición de recursos: Los hilos comparten los recursos de un proceso más fácilmente que los modelos de pase de mensajes o memoria compartida.
- Ahorro: El proceso de creación de los hilos es mucho más ágil que el de los procesos, y lo mismo ocurre con el cambio de contexto.
- Escalabilidad: El proceso multihilo puede aprovechar mejor las prestaciones de un mayor número de arquitecturas multiprocesador.
- Simplificación del código, lo que incrementa su eficiencia.

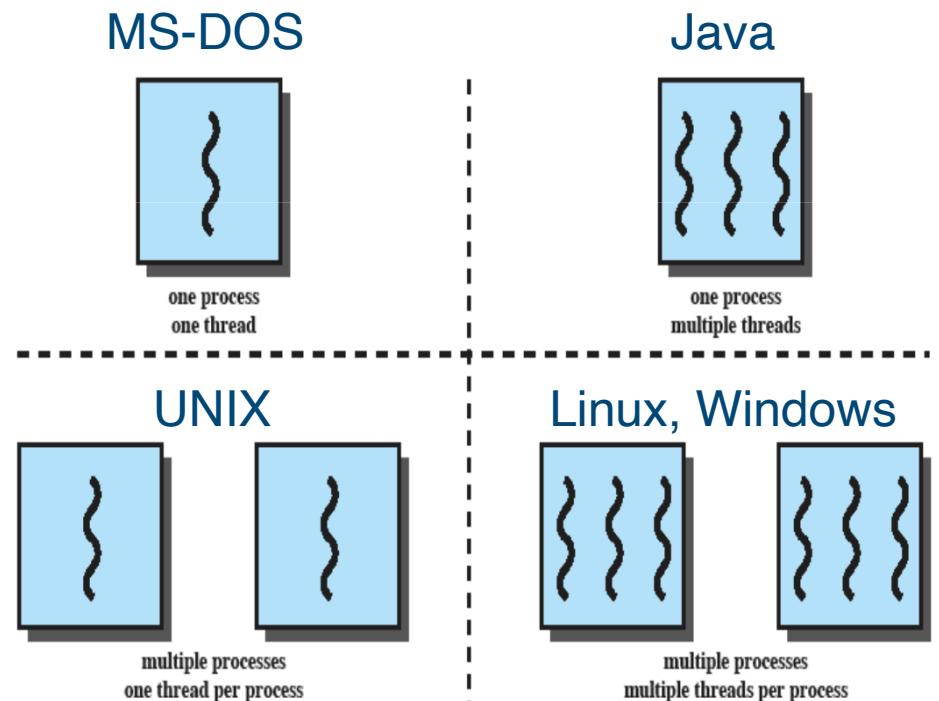
Entornos multihilo

● *Single-threaded:*

- Un único hilo de ejecución en un solo proceso.
 - Ej: MS-DOS.
- Un único hilo de ejecución en cada uno de los procesos.
 - Ej: UNIX.

● *Multi-threaded:*

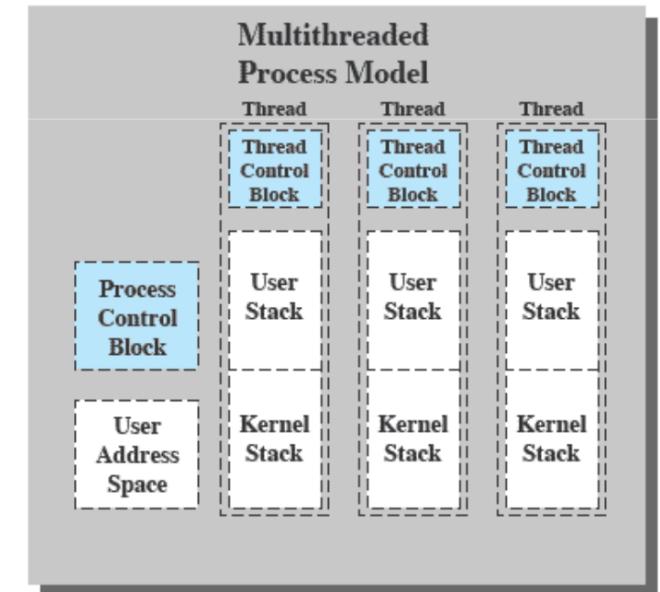
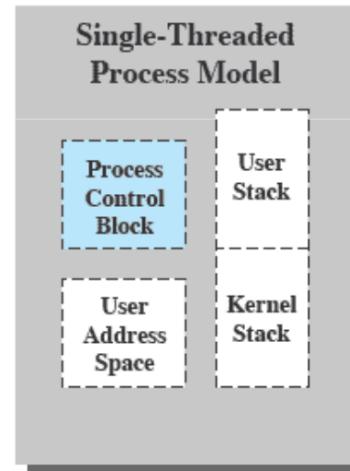
- Un proceso con múltiples hilos.
 - Ej: Java.
- Múltiples procesos con múltiples hilos.
 - Ej: Linux, Windows.



El bloque de control del hilo

- Como proceso, cada hilo tiene bloque de control propio, que agrupa la siguiente información:

- Thread ID.
- Contador de programa.
- Estado de la ejecución (Ready, Running, ...).
- Información sobre su planificación.
- Contexto almacenado (cuando no está usando la CPU).



Librerías para la creación de hilos

- Proporcionan al programador el API para crear y gestionar los hilos. Hay dos formas básicas de implementación:
 - La librería se ubica íntegramente en el espacio del usuario.
 - La librería se ubica a nivel de kernel apoyada por el S.O.
- Las librerías existentes son principalmente tres:
 - POSIX threads (Pthreads).
 - Win32 threads.
 - Java threads.
- En esta asignatura nos centramos en el uso de Pthreads:
 - Son un estándar ISO/IEEE y populares en muchos S.O. tipo UNIX.
 - El API especifica el comportamiento de la librería, pero la implementación depende del desarrollo de la librería.

Servicios proporcionados por Pthreads

- `int pthread_attr_init (pthread_attr_t *attr)`

- Permite inicializar los atributos de los hilos que se van a usar, tales como el tamaño de la pila, la prioridad o el algoritmo de planificación.

- `int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*func) (void *), void *arg)`

- Crea un hilo que ejecuta la función `func` con sus argumentos `arg` especificados en `attr`, y devuelve el nuevo threadID en `thread`.

- `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate)`

- Determina si un hilo es o no independiente.

Servicios proporcionados por Pthreads (2)

- **`int pthread_join (pthread_t thid, void *status)`**

- Suspende la ejecución del hilo hasta que acabe el hilo con ID thid.
- Devuelve el estado de terminación del hilo con ID thid.
- Es una forma de sincronización entre hilos.

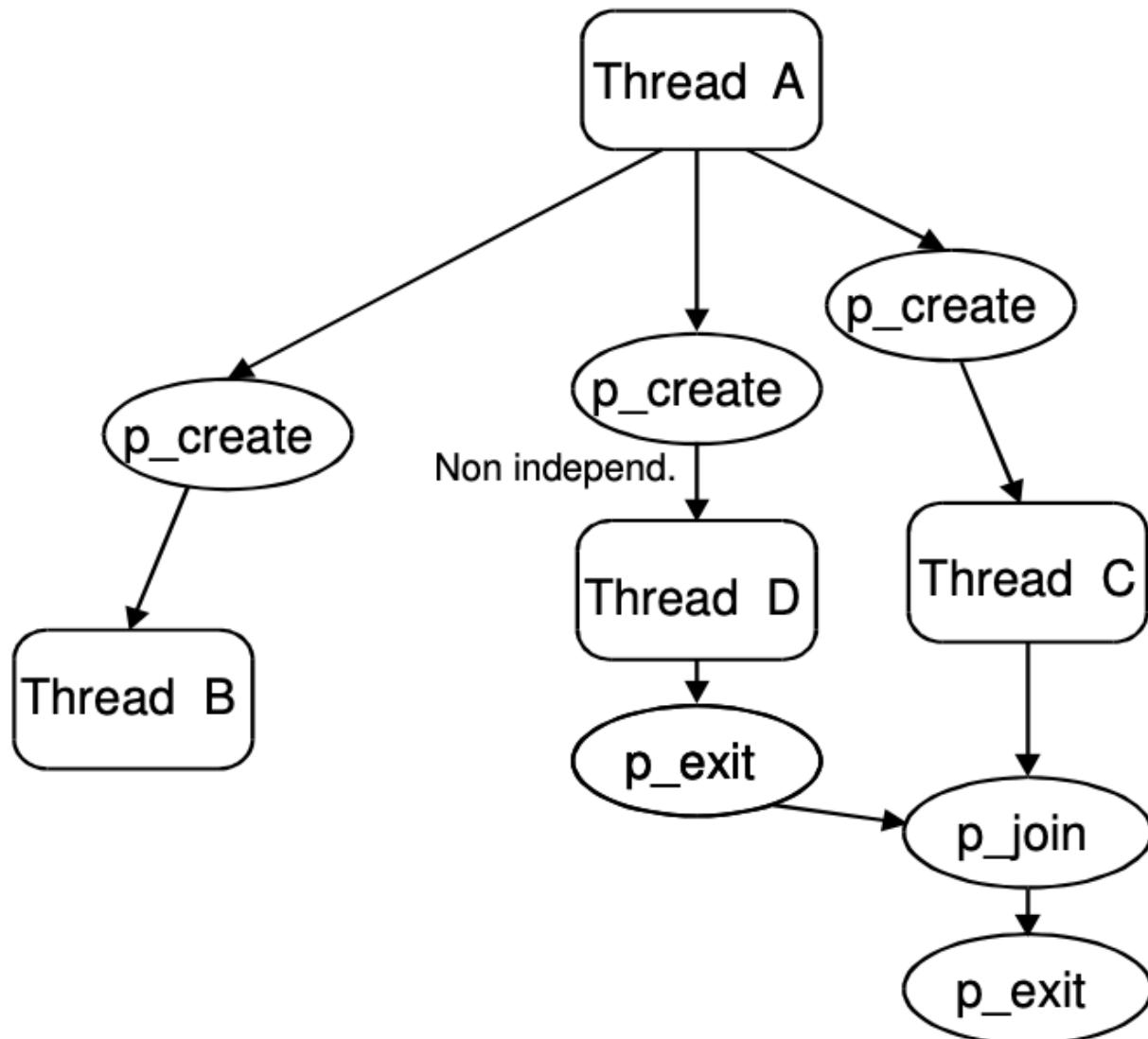
- **`int pthread_exit (void *status)`**

- Finaliza la ejecución de un hilo, devolviendo su estado de finalización a los hilos que se hayan unido previamente.

- **`pthread_t pthread_self (void)`**

- Devuelve el ID del hilo que llama a la función.

Ejemplo de jerarquía de hilos



Ejemplo de programa creado con Pthreads

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadID)
{
    printf("\n%d: Hello World!\n", threadID);
    pthread_exit(NULL);
}

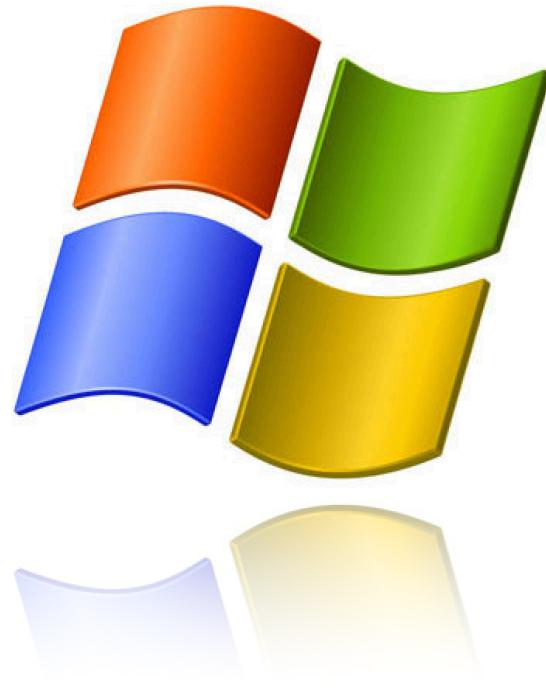
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for (t=0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR: Return code from pthread_create() is %d\n", rc);
            exit (-1);
        }
    }
    pthread_exit(NULL);
}
```

Una posible salida del programa (pero no la única) sería la siguiente:

```
Creating thread 0
Creating thread 1
0: Hello World!
1: Hello World!
Creating thread 2
Creating thread 3
2: Hello World!
3: Hello World!
Creating thread 4
4: Hello World!
```

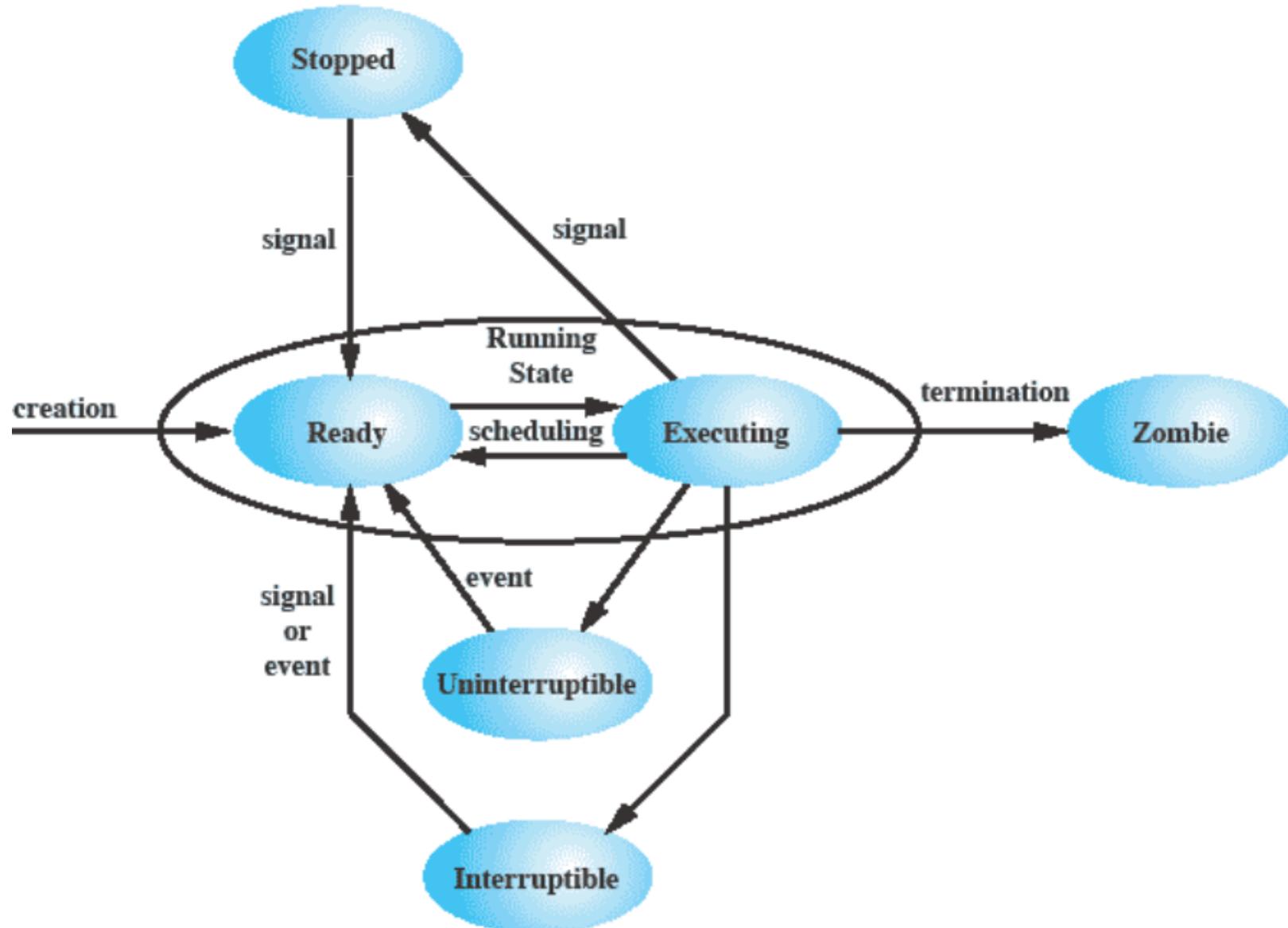
La API Win32 Thread (Windows)

- **BOOL CreateThread (LPSECURITY_ATTRIBUTES lpsa, DWORD cbsStack, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpvThreadParam, DWORD fdwCreate, LPDWORD lpIdThread);**
- Crea un hilo.
- **VOID ExitThread (DWORD dwExitCode);**
- Finaliza la ejecución del hilo.



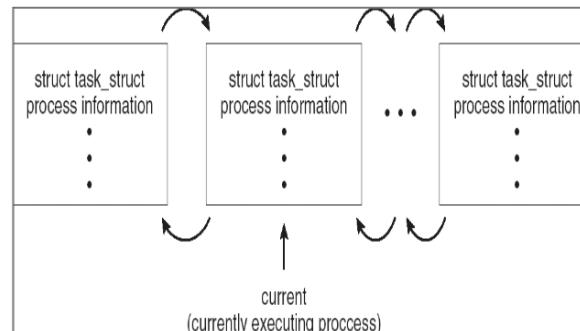
III. Ejemplos en los S.O. actuales

Estados de los procesos y los hilos en Linux



Procesos activos en Linux

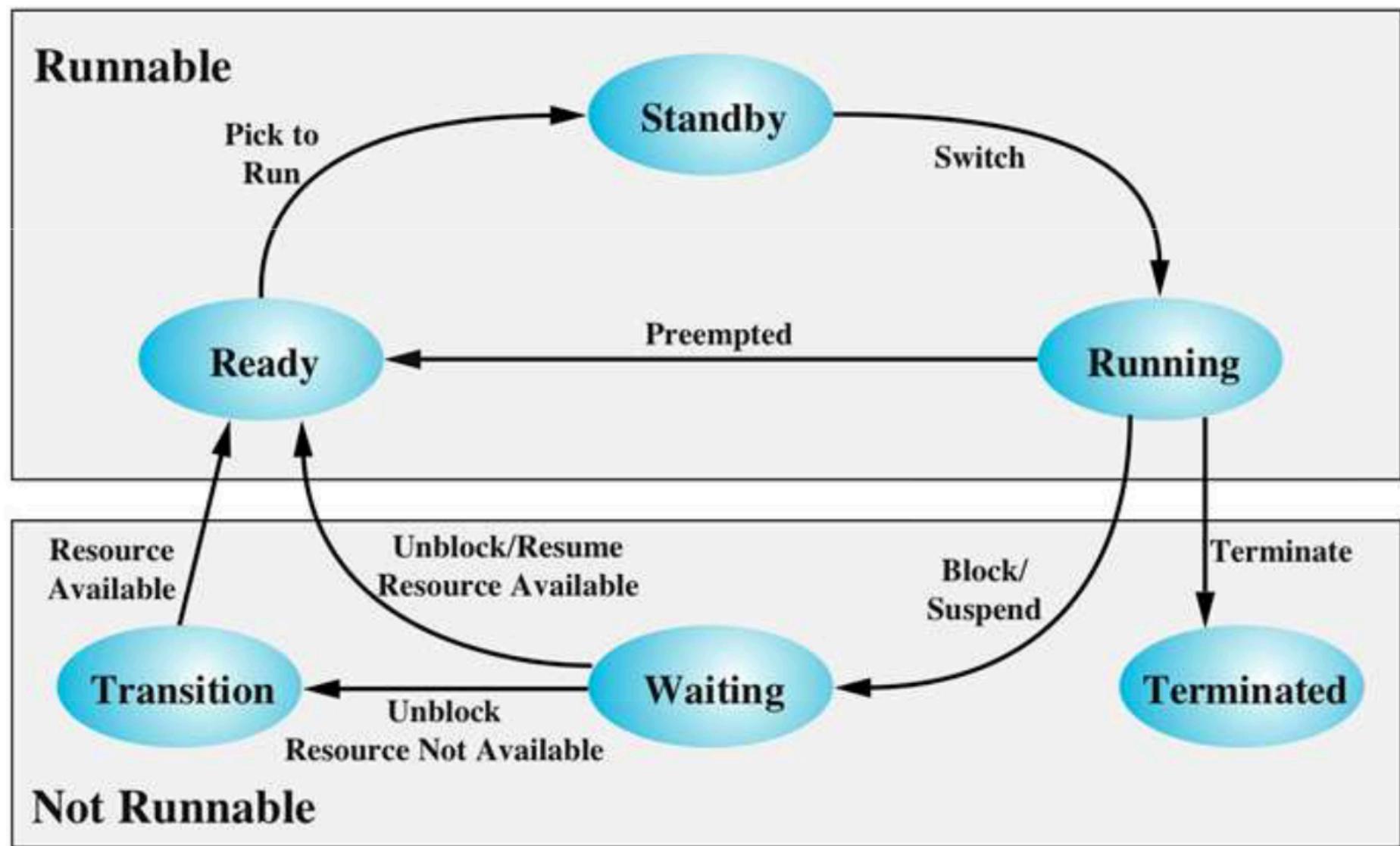
- El PCB (Process Control Block) en Linux se representa por la estructura `task_struct`, que agrupa la siguiente info:
 - El estado del proceso.
 - Información acerca de la planificación y la gestión de memoria.
 - La lista de ficheros abiertos.
 - Punteros al padre del proceso y todos sus hijos.
- Todos los procesos activos se representan con una lista doblemente enlazada de `task_struct`, y el kernel mantiene un puntero al proceso actualmente en ejecución:



Hilos en Linux

- En Linux, los hilos se denominan *tasks* (tareas).
- El hilo se crea con la llamada al sistema `clone()`, que permite a una tarea hija compartir el espacio de direcciones de su tarea padre.
- Los *flags* controlan el comportamiento de estas tareas.
 - `CLONE_FS`: Comparten la información del sistema de ficheros.
 - `CLONE_VM`: Comparten el mismo espacio de direcciones.
 - `CLONE_SIGHAND`: Comparten los manejadores de señales.
 - `CLONE_FILES`: Comparten el conjunto de ficheros abiertos.

Estados de los procesos y los hilos en Windows





Objetos de los procesos y los hilos en Windows

Object Type

Process

Process ID
Security Descriptor
Base priority
Default processor affinity
Quota limits
Execution time
I/O counters
VM operation counters
Exception/debugging ports
Exit status

Object Body Attributes

Create process
Open process
Query process information
Set process information
Current process
Terminate process

Services

(a) Process object

Object Type

Object Body Attributes

Thread ID
Thread context
Dynamic priority
Base priority
Thread processor affinity
Thread execution time
Alert status
Suspension count
Impersonation token
Termination port
Thread exit status

Services

Create thread
Open thread
Query thread information
Set thread information
Current thread
Terminate thread
Get context
Set context
Suspend
Resume
Alert thread
Test thread alert
Register termination port

Thread

(b) Thread object