

S.O. Tema 2.1: Procesos

1. Procesos

1.1. 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**:

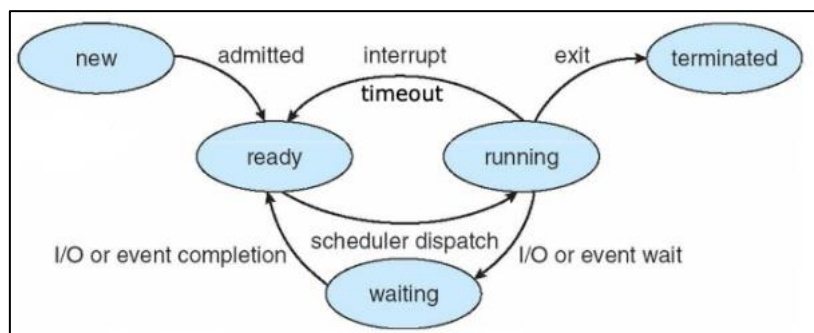
- **Memoria** para **alojar** el **programa**.
- **Registros** de la **CPU** para **valores** de la **ejecución** (PC, AX, BX, ...).
- La **pila**, para **datos temporales**, **paso** de **parámetros** a **funciones** y **direcciones** de **retorno** de ellas.

1.2. Estado 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.

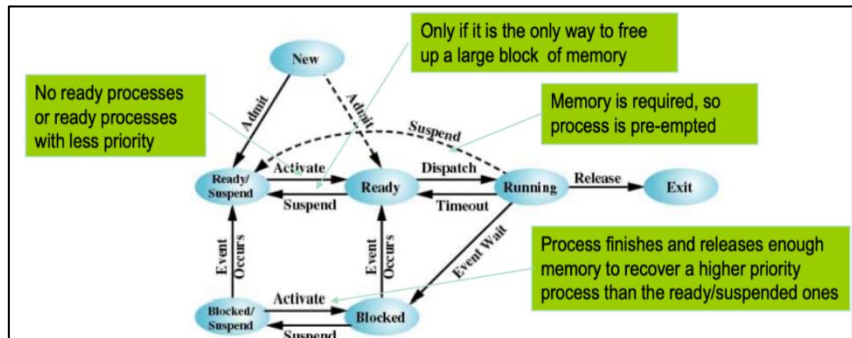


1.3. Suspensión de procesos

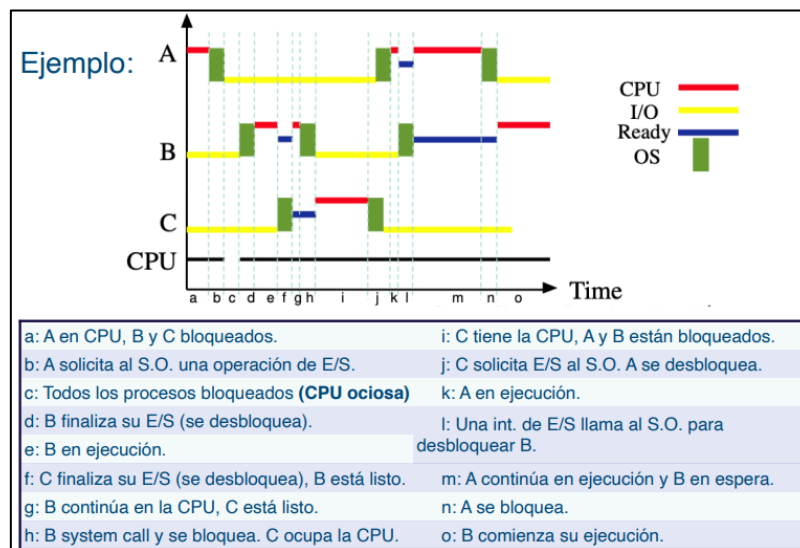
-En los **sistemas** de **memoria virtual**, los **procesos** pueden **trasladarse temporalmente** a **disco** para **liberar** su **memoria**, lo que debe reflejarse contemplando **dos nuevos estados**:

- **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).



1.4. Seguimiento de los procesos



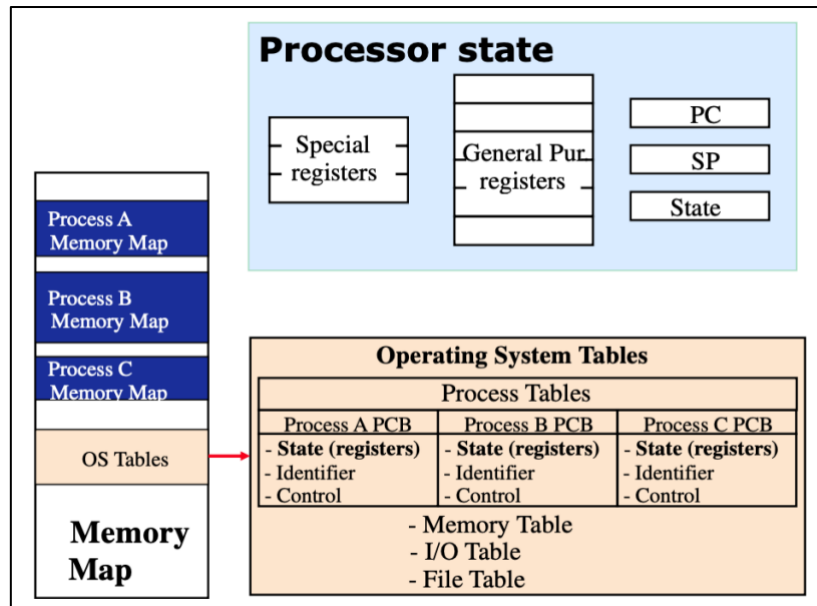
1.5. El bloque de control del proceso (PCB)

-Cuando la **CPU conmuta** del **proceso A** al **B** necesita **alojar** cierta **información** de **A** para **retomar**lo 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...

-En el **siguiente diagrama** se muestran los **siguientes registros** de los **PCB** por parte del **S.O.**



1.6. Imagen de un proceso

-Delimita el **conjunto** de **información** que lo **caracteriza**.

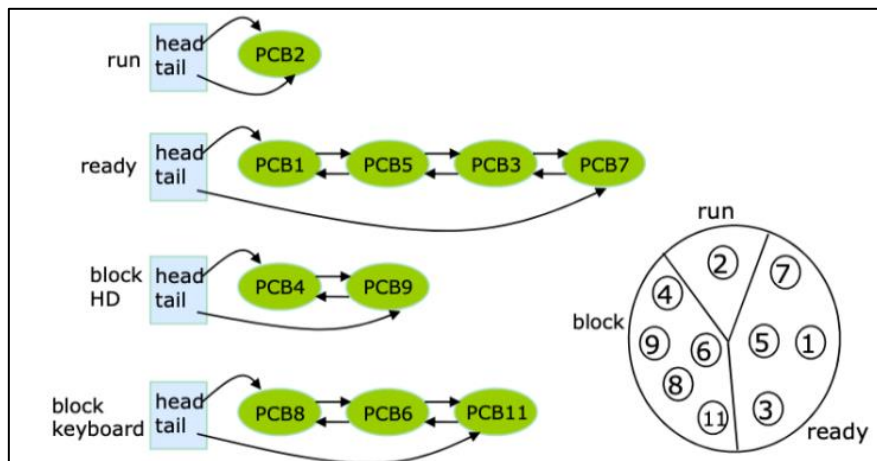
-Se **compone** de **tres elementos** principales:

- 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**.

1.7. 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**.

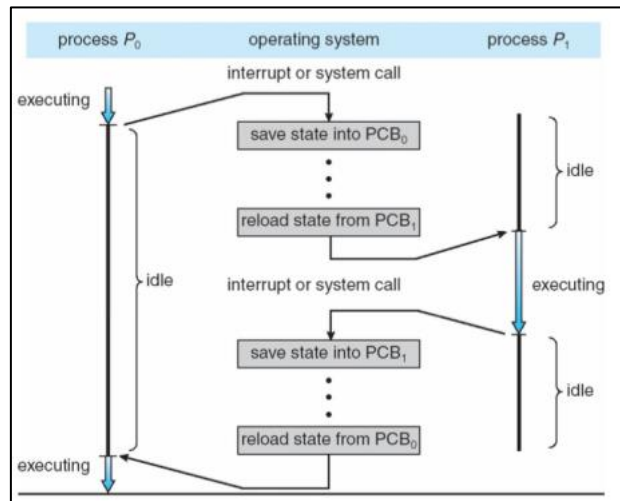


1.8. Conmutación de la CPU de un proceso a otro

-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**.

-En la **imagen** vemos como se **está ejecutando** el proceso **P₀**, se **produce** una **interrupción** o **llamada** al **sistema** y hay un **tiempo** (idle) en el que se **modifica** el **PCB**.



1.9. 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**:

1.10. 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.

-El **proceso inicial** se denomina "Init", que crea **daemons**, entre ellos el **shell**, desde el que el usuario **lanza nuevos procesos**.

1.11. Creación de procesos en UNIX con 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).

1.12. 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).

-Ejemplo de uso (importante!!):

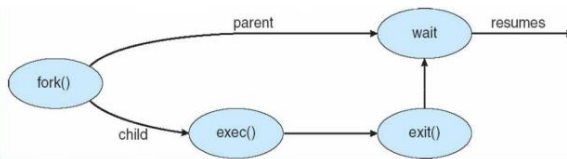
```

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

int main()
{
    pid_t pid;

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

```



-Mejora de lo anterior, **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;

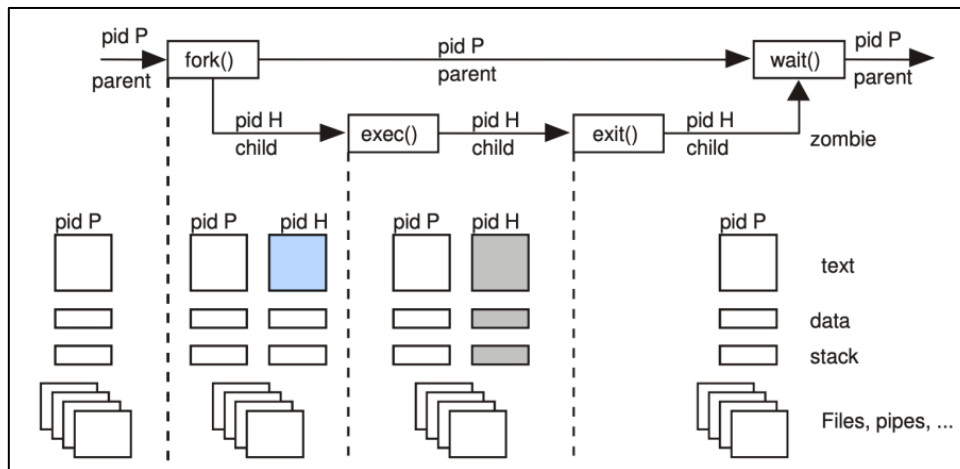
    pid = fork();
    if (pid == 0) {
        /* child process */
        execlp("/bin/ls", "ls", "-l", NULL);
        exit(-1);
    }
    else {
        /* parent process: will wait only for */
        while (pid != wait(&status)); /* that particular child to end */
        printf("Child finished");
  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

```

1.13. 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**:



1.14. 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**.

1.15. Señales

-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).`

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

1.16. 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.

1.17. 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**:

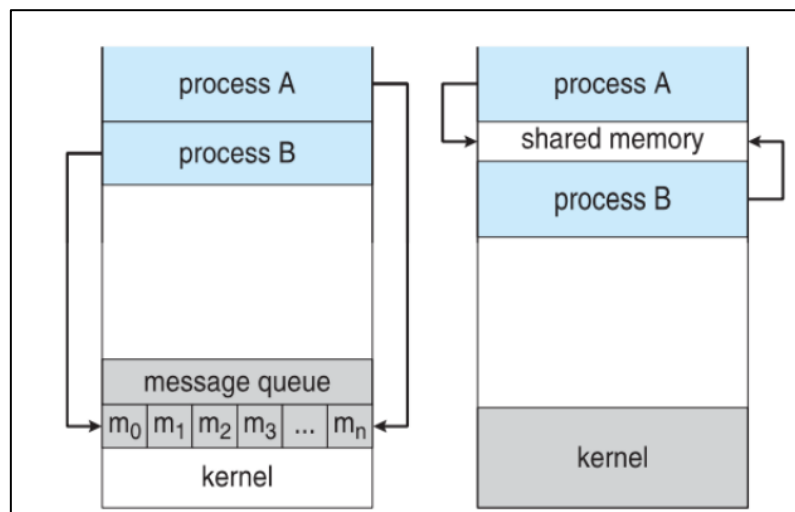
- **Memoria compartida.**
- **Pase de mensajes.**

-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.



2. Hilos

2.1. Concepto, definición e información asociada

-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.

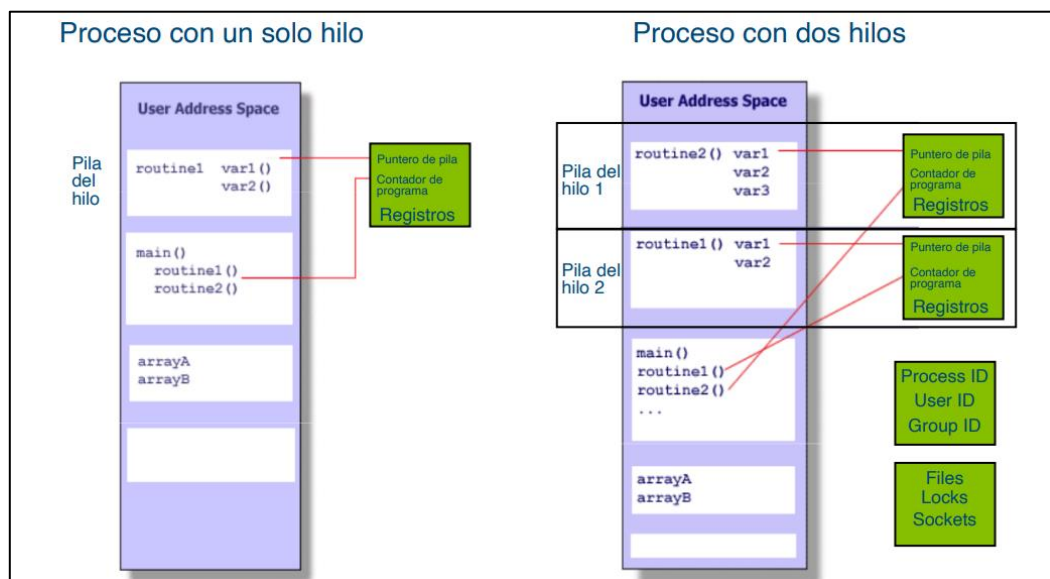
-El **proceso retiene** la **propiedad** de los **recursos** (memoria, ficheros, periféricos), mientras que sus **hilos despliegan** el **paralelismo** entre los cores para acelerar el proceso, y son los elegidos por el planificador de cada uno de estos cores.

-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, etc)
 - Los **procesos hijo**, las **variables globales** y las **señales**, entre otras cosas.

2.2. Espacio de direcciones para los hilos



2.3. 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.

2.4. 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.

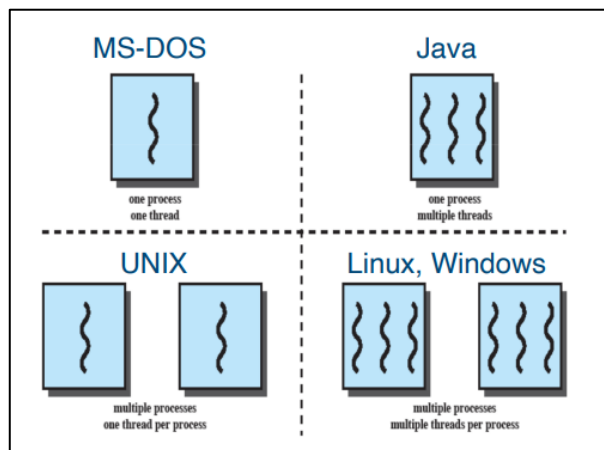
2.5. Entornos multihilo

-**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, etc.



2.6. 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).

2.7. 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 y Java threads.

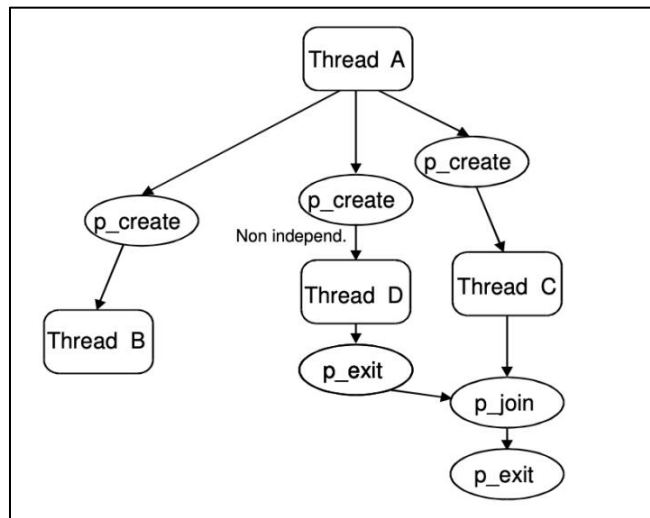
-En esta **asignatura** nos **centraremos** en el uso de **Pthreads**:

- 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.

2.8. Servicios proporcionados por Pthreads

- **int pthread_attr_t 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_t setdetachstate (pthread_attr_t *attr, int detachstate)**: determina si un hilo es o no independiente.
- **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.

2.9. Ejemplo de jerarquía de hilos



2.10. Ejemplo de programa creado con Pthreads

3. Ejemplos en los S.O. actuales

```

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

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

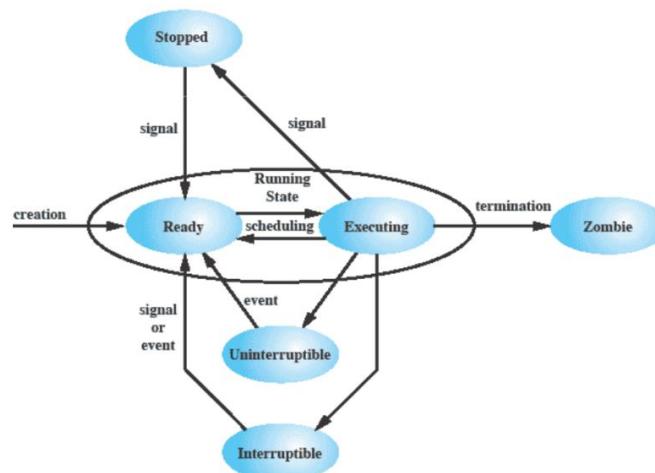
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for (t=0; t < NUM_THREADS; t++)
    {
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR: Return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

3.1. Estados de los procesos y los hilos en Linux



3.2. Procesos activos en Linux

-El **PCB (Process Control Block)** en Linux se representa por la estructura `task_struct`, que agrupa la siguiente información:

- 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.

3.3. Hilos en Linux

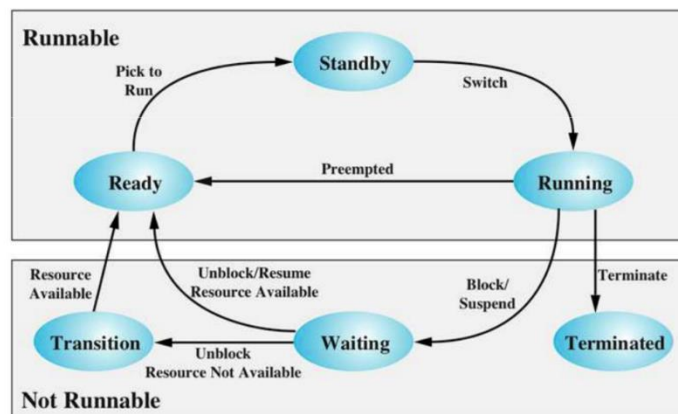
-En **Linux**, los **hilos** se denominan **tasks** (tareas).

-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.

3.4. Estados de los procesos y los hilos en Windows



3.5. Objetos de los procesos y los hilos en Windows

