# UT1 – Programación Multiproceso


**2ºDAM - Programación Servicios y Procesos**

**Alfonso Rebolleda Sánchez – arebolleda@educa.madrid.org**

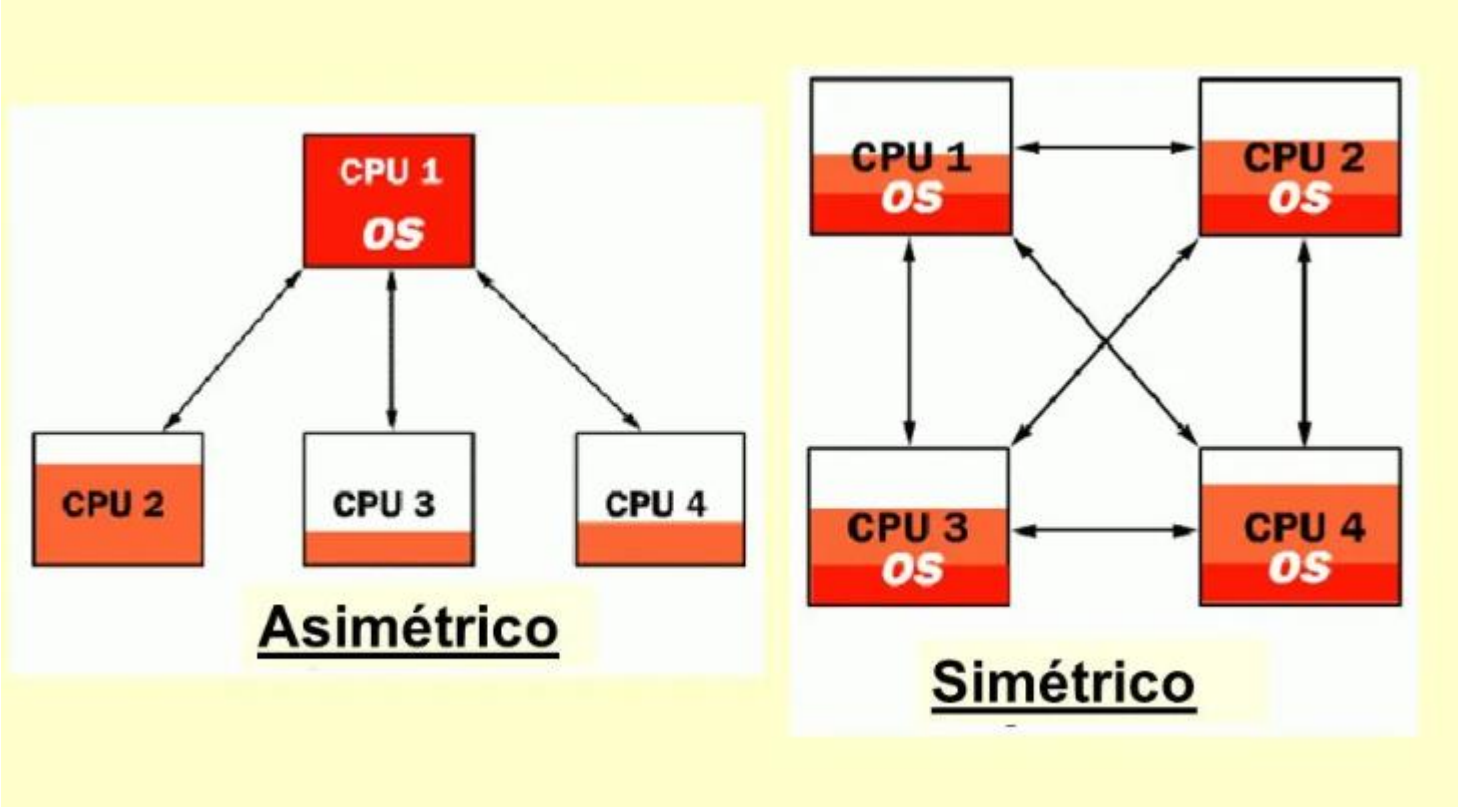
# Objetivos

- Conocer las características de un proceso y su ejecución por sistema operativo.
- Conocer las características y diferencias programación concurrente, paralela y distribuida.
- Crear procesos en Linux y utilizar clases Java para crear procesos
- Desarrollar programas que ejecuten tareas en paralelo

# Introducción

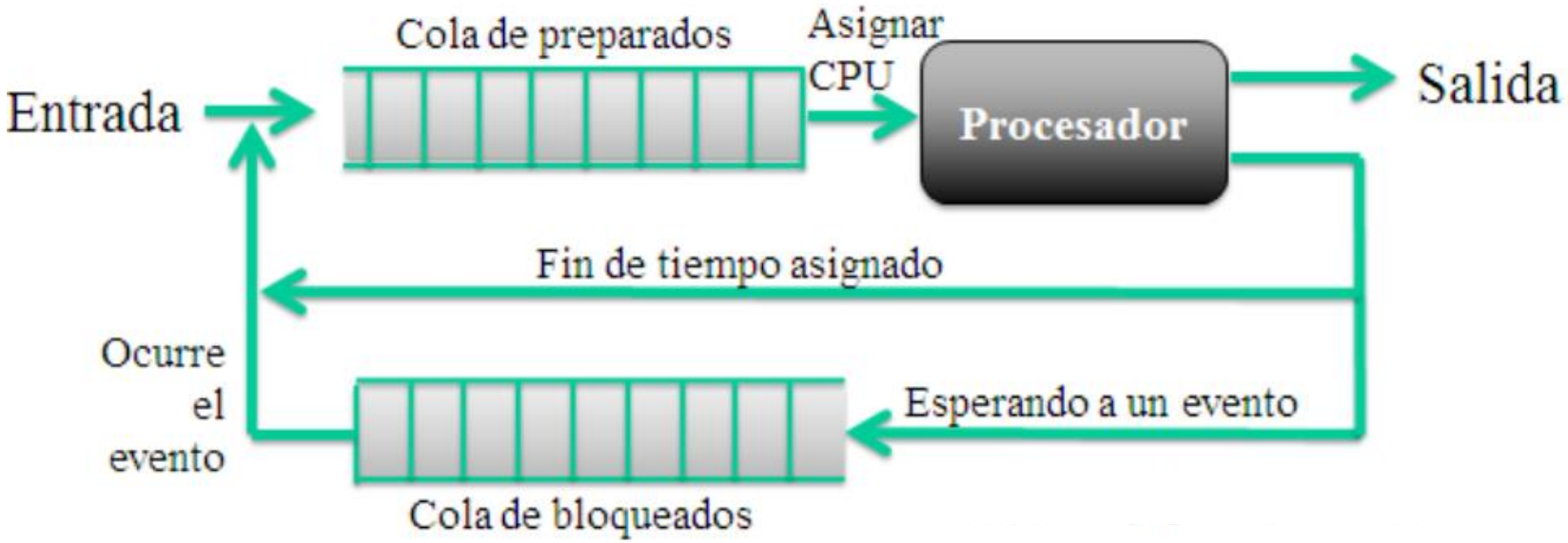
**Sistema Operativo Multiproceso** permite ejecutar más de un proceso a la vez.

La única forma de ejecutar varios procesos simultáneamente es tener varias CPUs.



# 1. Introducción

En SO con una única CPU se va alternando la ejecución de los procesos, es decir, se desaloja un proceso de la CPU, se ejecuta otro y se vuelve a planificar el primero; esta operación se realiza tan rápido parece cada proceso tiene dedicación exclusiva.



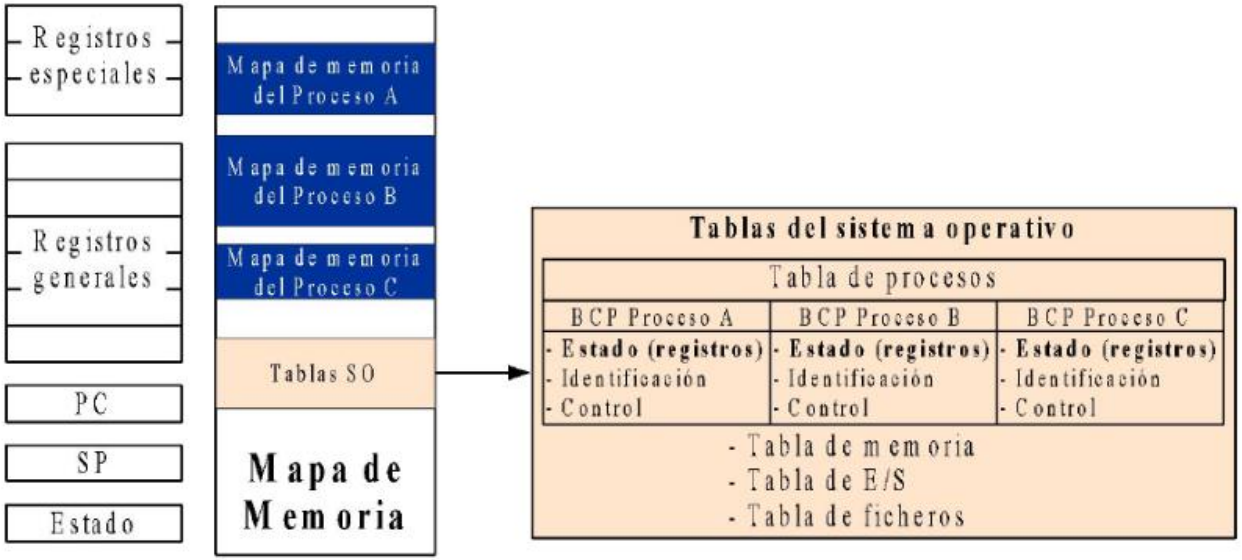

La **programación multiproceso** múltiples procesos puedan estar ejecutándose simultáneamente sobre el mismo código de programa. Desde una misma aplicación podemos realizar varias tareas de forma simultánea, o lo que es lo mismo, podemos dividir un proceso en varios subprocesos.

# 2. Procesos

Un **proceso** es un **programa en ejecución**

Se compone:

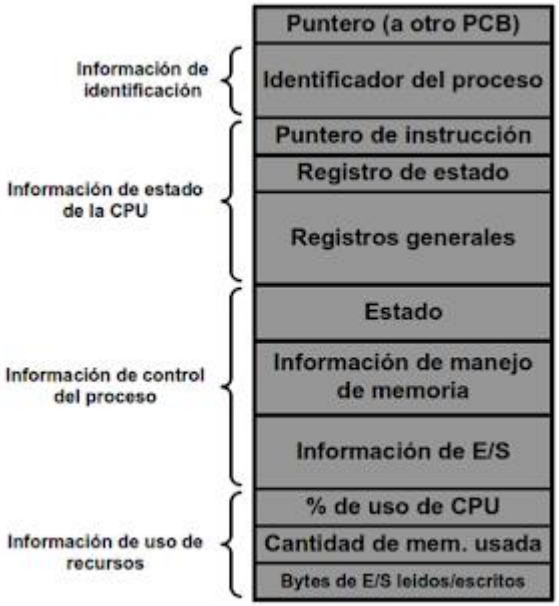
- ✓ Ejecutable del programa
- ✓ Los datos y la pila del programa,
- ✓ El contador de programa,
- ✓ El puntero de pila y otros registros
- ✓ resto información necesaria para ejecutar el programa



# 2. Procesos

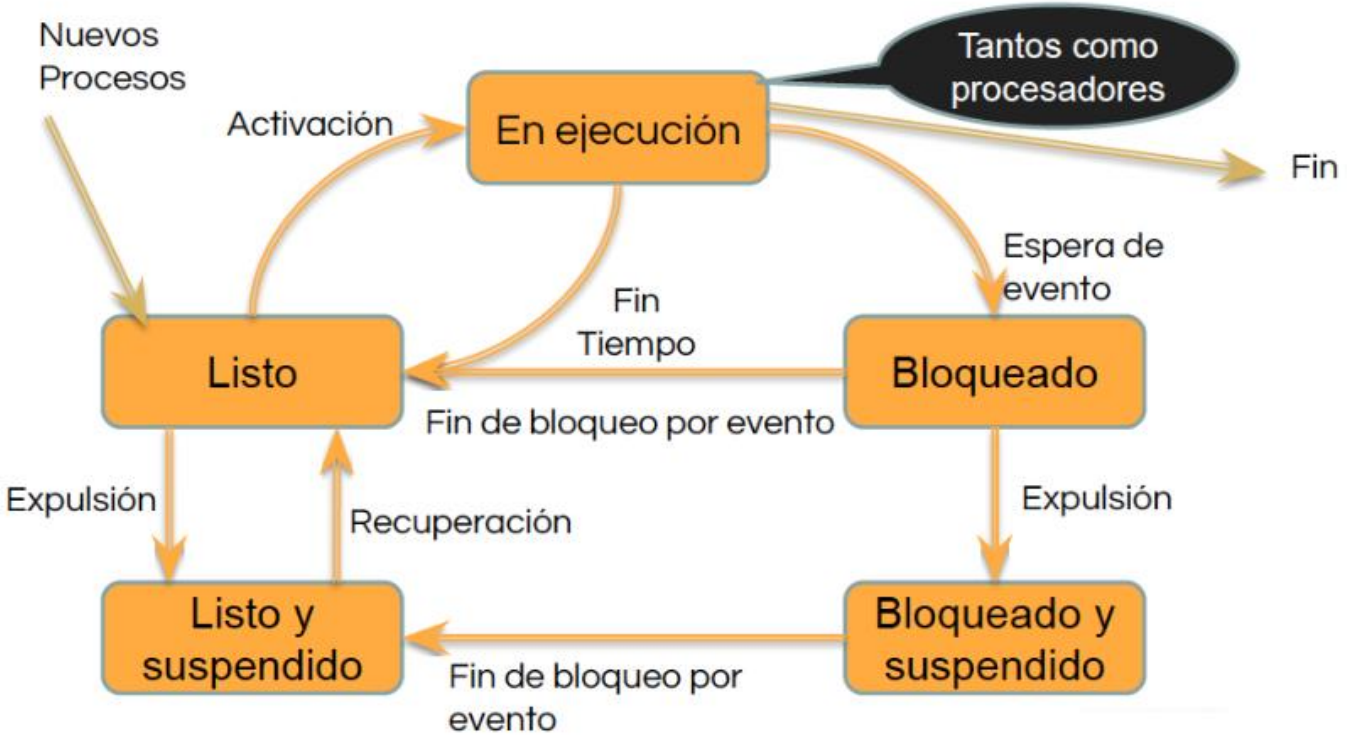
- Cada proceso se representa en el SO, por un conjunto de datos, que incluye toda la información necesaria para definirlo: el estado, recursos utilizados, registros.

- Este conjunto de datos se conoce como **Bloque de Control de Procesos (PCB)**, y es toda la información que el S.O. necesita para ejecutar el programa.



# 2.1. Estados Proceso

Un **proceso** es pasa por diferentes estados a lo largo de su vida



# 2.2. Planificación de procesos

- El objetivo multiproceso es maximizar la utilización del procesador teniendo múltiples procesos ejecutando de forma solapada: CPU compartida en tiempo.

- Para esto, el **proceso planificador** (*process scheduler)* selecciona un proceso disponible para que ejecute en la CPU.

- En sistemas con una única CPU nunca hay más de un proceso en ejecución. El resto esperan hasta que la CPU queda libre y pueda ser planificada.

- En sistemas con múltiples procesos, la planificación utiliza colas de procesos que los agrupan por estados:

  **Cola de trabajo** (todos los procesos en el sistema)
  **Cola de preparados** (los que están en memoria principal, listos y esperando a ejecutar)
  **Cola de dispositivos** (los que esperan un dispositivo de E/S)
  **Proceso de migración entre varias colas**

# 2.3. Control procesos

Se pueden ejecutar comandos del SO con funciones de C

- Función ***system()*** se encuentra en la librería estándar **stdlib.h**

  ***int system(const char *cadena)*** recibe como parámetro una cadena de caracteres que indica el comando que se desea procesar. Devuelve -1 si ocurre un error y el estado devuelto por el comando en caso contrario

- Función **execl** se encuentra en la librería estándar **unistd.h**

  **int execl(const char *fichero, const char *arg0, … , char *argn, (char *)NULL)** recibe el nombre del fichero que se va a ejecutar y luego los argumentos terminando con un puntero nulo. Devuelve -1 si ocurre algún error y en la variable global **errno** se pondrá el código de error adecuado.

- Función **execv** se encuentra en la librería estándar **unistd.h**

  **int execv(const char *path, char *const argv[]);**
  recibe el nombre del fichero que se va a ejecutar y luego una lista de argumentos. Devuelve -1 si ocurre algún error y en la variable global **errno** se pondrá el código de error adecuado.

# 2.3. Control procesos

- Función ***system()*** crea un proceso hijo para ejecutar el comando que se le pasa como argumento

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

int main()
{
        printf("Inicio del programa\n");
        system("ps -l");
        printf("Fin del programa\n");
}
```

- Funciones **execl/execv** reemplazan el código del proceso actual por el código del comando a ejecutar

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

int main()
{
        printf("Inicio del programa\n");
        execl("/bin/ps","ps","-l",NULL);
        printf("Fin del programa\n");
}
```

# 2.3. Operaciones sobre procesos

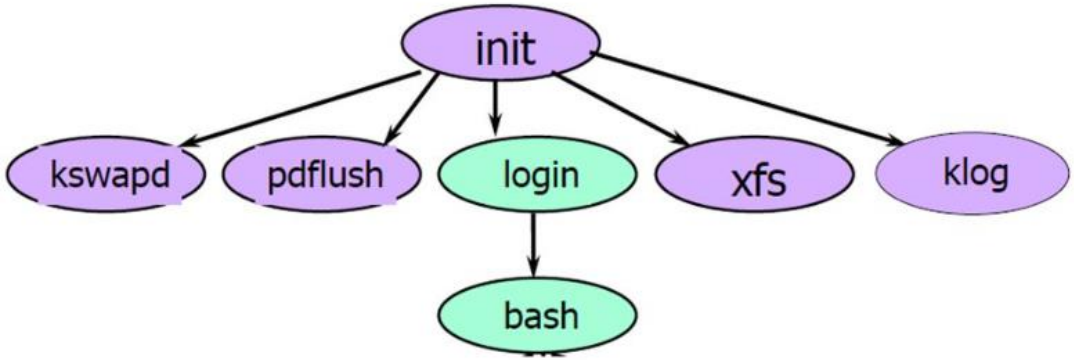
Las operaciones básicas sobre procesos son:

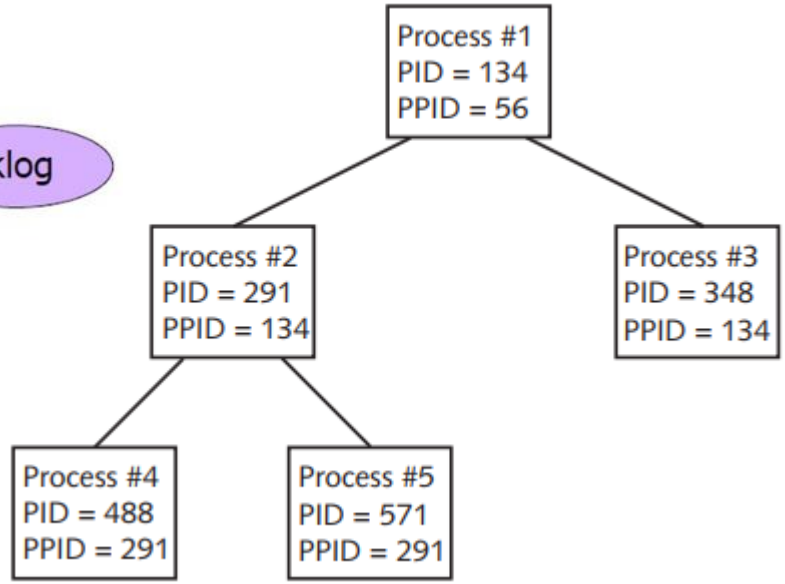
1. Creación
2. Cambio de Contexto
3. Terminación

# 2.3. Operaciones sobre procesos

## 1. Creación procesos

- Los SO permiten la creación y borrado de procesos dinámicamente.
- Los procesos padre crean procesos hijo que, a su vez, pueden crear otros procesos; se forma un árbol de procesos.



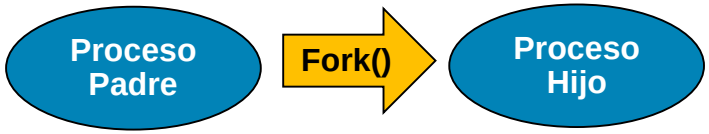



# 2.3. Operaciones sobre procesos

## 1. Creación procesos - Comandos

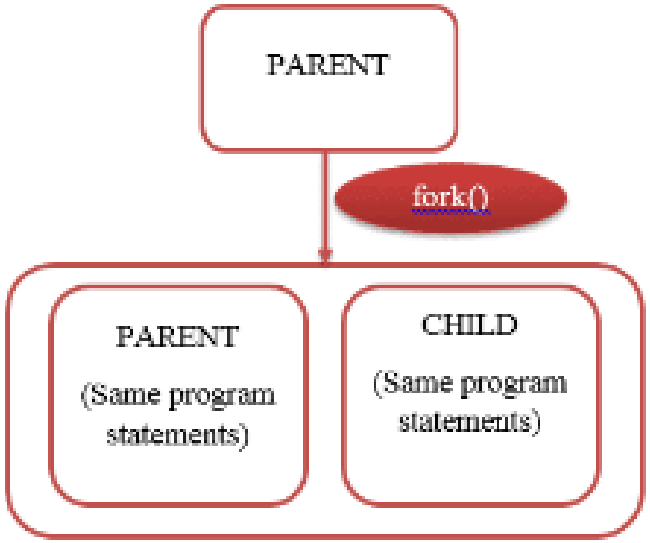
### UNIX/LINUX

**`fork`**: crea un nuevo proceso




### Windows

**`CreateProcess()`** permite crear un nuevo proceso pasándole directamente el programa que va a ejecutar.



# 2.3. Operaciones sobre procesos

## 2. Cambio contexto procesos

- El **proceso** que está ejecutando en la **CPU la abandona**
- Otro proceso con estado **preparado** se mueve al estado **ejecutando**
- Las acciones que suceden para cambiar el contexto son:
  Salvar el contexto actual (contador de programa, puntero de pila, …)
  Actualizar PCB del proceso que sale de la CPU e insertarlo en la cola adecuada
  Actualizar PCB del proceso que va a ejecutar
  Restaurar el nuevo contexto
- El tiempo de cambio de contexto es un **gasto** en recursos y por tanto en tiempo de ejecución

# 2.3. Operaciones sobre procesos

## 3. Terminación procesos

Un **proceso** que está ejecutando en la **CPU termina o finaliza:**

- El proceso ejecuta la última instrucción
- El **padre puede terminar la ejecución del proceso hijo**
  - El hijo ha excedido los recursos reservados
  - La tarea asignada al hijo ya no es requerida
  - El padre termina y el sistema operativo no permite al hijo continuar si su padre termina. Este efecto se conoce como **terminación en cascada**
- El proceso recibe una señal del sistema para su finalización

# 2.4. Comunicación entre procesos

**IPC** comunicación entre procesos (Inter-Process Communication o IPC)

Mecanismos que permiten procesos **comunicarse y sincronizarse** entre ellos.

La comunicación entre procesos puede estar motivada por la competencia o el uso de recursos compartidos o porque varios procesos deban ejecutarse sincronizadamente para completar una tarea.

Diferentes mecanismos:

- ✓ Pipes
- ✓ Colas de mensajes
- ✓ Semáforos
- ✓ Segmentos de memoria compartida

# 2.4.1. Pipes

**Pipes** comunican 2 procesos mediante mecanismo half-dúplex (provee un canal de comunicación unidireccional entre dos procesos que cooperan)


Un pipe es una especie de **falso fichero** que permite conectar 2 procesos.

Si el proceso P1 quiere enviar datos al proceso P2, los escribe en el pipe como si fuera un fichero de salida. El proceso P2 leerá los datos del pipe como si fuera un fichero de entrada.

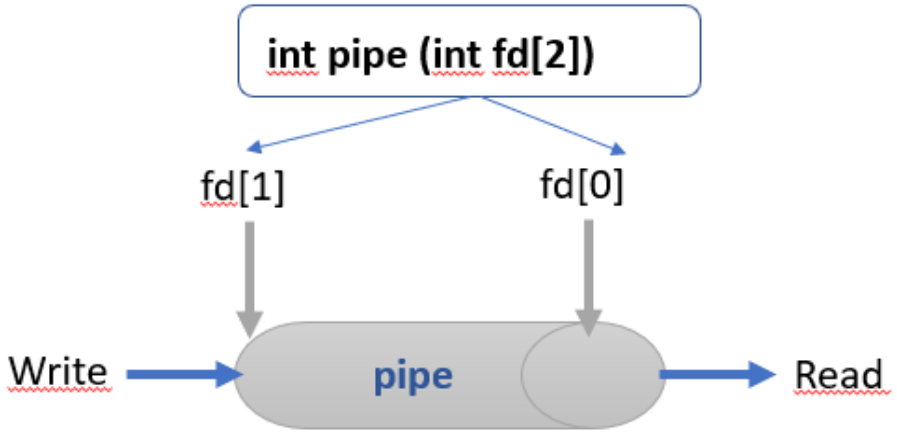
Cuando proceso quiere leer del pipe y está vacío -> se bloquea a la espera datos

Cuando proceso quiere escribir en pipe y está lleno -> se bloquea a la espera se vacíe

# 2.4.1. Pipes

Función **pipe** permite crear el pipe:

- Argumento: array 2 enteros
- filedes[0]: descriptor lectura
- filedes[1]: descriptor escritura
- Devuelve 0 si éxito, -1 si error




Función **read** permite leer datos del pipe:

***int read( int fd, void *buf, int count);***

intenta leer **count** bytes del descriptor de fichero **fd**, para guardarlos en buffer **buf**. Devuelve el número de bytes leídos; si comparamos este valor con la variable **count** podemos saber si ha conseguido leer tantos bytes como se pedían.

Función **write** permite escribir datos en el pipe:

***int write( int fd, void *buf,int count);***

write() es muy similar, en **buf** especificamos los datos a escribir, definimos su tamaño en **count** y especificamos el fichero en el que escribiremos en **fd**

# 2.4.2. Named Pipes o FIFOS (First In First Out)

Permiten comunicar procesos no emparentados

Se implementa como **fichero con nombre** que existe en sistema de ficheros y que pueden abrir, leer y escribir múltiples procesos.

Los datos escritos se leen como en una cola, primero en entrar (FIRST IN), primero en salir (FIRST OUT); y una vez leídos no pueden ser leídos de nuevo.

Para utilizarlos:

- ✓ Se crean con la función **mkfifo**: ***int mkfifo(const char *pathname, mode_t mode);***
- ✓ Se usan las funciones ***read()*** y ***write()*** de ficheros
- ✓ Si se abre el fichero para lectura (***O_RDONLY***), el fichero se bloqueará para las escrituras
- ✓ Si se abre el fichero para escritura (***O_WRONLY***), el fichero se bloqueará hasta que otro proceso abra el named pipe para leer

# 2.4.3. Señales

Una señal es un "aviso" que puede enviar un proceso a otro proceso. Las señales **permiten la comunicación y sincronización de procesos**, aunque resultan un mecanismo más pobre que los pipes, ya que **no permiten transmitir datos**

Las señales pueden ser generadas por:

- El **kernel sistema operativo**: cuando se detecta un error de software o de hardware en la ejecución del proceso.
- El **usuario**: al pulsar ciertas teclas (Ctrl+Z o Ctrl+C) en un terminal, y también a través de la línea de comando (***comando kill*** permite que el usuario envíe señales a un proceso)
- Un **proceso**: envía señal otro proceso/s utilizando ***llamada al sistema kill o killpg***

Como las señales pueden aparecer en cualquier instante, el proceso debe indicar al kernel qué es lo que ha de hacer cuando recibe una señal determinada.

El kernel puede actuar de tres formas diferentes:

1) Ignorar la señal
2) Capturar la señal
3) Ejecutar la rutina/acción por defecto

# 2.4.3. Señales

**1) Ignorar la señal**

Mediante el establecimiento de la constante SIG_IGN, el proceso ignorará la recepción de la señal.

Esto implica que el proceso no realizará ninguna acción al recibirla; su ejecución continuará exactamente en el mismo punto donde se encontraba.

Cuando se ignora una señal, suele decirse que el proceso se hace inmune a ella. Esta inmunidad no significa que el proceso no quede afectado por la causa que generó la señal, sino que simplemente la ignora.

Por otra parte, un proceso que ignorara todas las señales no podría ser gestionado por el administrador, ya que ésta no respondería a sus instrucciones: por ejemplo, si proceso bucle infinito y su ejecución no terminara nunca → **señales no pueden ignorarse**

# 2.4.3. Señales

**2) Capturar la señal**

El programador del proceso puede preparar una rutina que se ejecute al recibirse una señal determinada.

El código de la rutina debe desarrollarse teniendo en cuenta que esta rutina no se va a ejecutar en el momento deseado por el programador, sino en cualquier instante en que se produzca un evento.

El código deberá, asimismo, responder a lo que ocurre en el sistema.

Cuandose termina la ejecución de esta rutina, el desarrollo del proceso puede continuar en elpunto en que se había interrumpido

# 2.4.3. Señales

**3) Ejecutar acción por defecto**

Mediante la especificación de la constante SIG_DFL, el kernel llama a una función determinada cada vez que la señal reaparece.

Cada señal tiene asignada una acción por defecto. Suelen ser:

- ***exit*** → el proceso receptor de la señal finaliza
- ***core*** → el proceso receptor finaliza y deja un fichero con imagen de memoria del contexto del proceso
- ***stop*** → el proceso receptor se detiene

# 2.4.3. Señales

**¿Qué señales existen?**

Cada señal posee un nombre que comienza por SIG, mientras que el resto de los caracteres se relacionan con el tipo de evento que representa. Asimismo, cada señal lleva asociado un número entero positivo, que es el que intercambia con el proceso cuando éste recibe la señal.

```
alumno@alumnomv:~/psp/ut1$ kill -L
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

# 2.4.3. Señales

| Tipo | N.º | Significado | Acción por defecto |
|---|---|---|---|
| *SIGHUP* | 1 | **Colgado**. Se envía al controlador de terminales cuando se detecta la desconexión de algún terminal, o cuando el proceso principal de un grupo termina. En este caso, se envía a todos los procesos miembros del grupo. Se suele emplear para notificar a procesos demonio que deben realizar una relectura de sus ficheros de configuración; se emplea esta señal, ya que este tipo de procesos no suele tener asociado ningún terminal, y no es probable que reciban esta señal por ningún otro motivo. | Fin del proceso que la recibe |
| *SIGINT* | 2 | **Interrupción**. Se envía a todos los procesos asociados con un terminal cuando se pulsa la tecla de interrupción (**Ctrl + C**). Se suele emplear para terminar la ejecución de programas que están generando mucha información no deseada por pantalla. | Fin del proceso que la recibe |

| Tipo | N.º | Significado | Acción por defecto |
|---|---|---|---|
| *SIGQUIT* | 3 | **Abandonar**. Se envía a todos los procesos asociados con un terminal cuando se pulsa la tecla de salida. Es muy similar a *SIGINT*, ya que su acción por defecto es la misma, pero, además, genera un fichero *core*. | Fin del proceso que la recibe y generación de fichero *core* |
| *SIGILL* | 4 | **Instrucción ilegal**. Se envía a un proceso cuando el hardware detecta que éste ha ejecutado una instrucción ilegal. | Fin del proceso que la recibe y generación de fichero *core* |

# 2.4.3. Señales

| Tipo | N.º | Significado | Acción por defecto |
|---|---|---|---|
| *SIGKILL* | 9 | **Eliminar**. Es una de las señales que no puede ser ignorada. Proporciona al administrador del sistema un medio seguro de terminar con cualquier proceso. | Fin del proceso que la recibe y generación de fichero *core* |
| *SIGUSR1* | 10 | **Señal de usuario 1**. Es una de las dos señales reservadas al usuario para su empleo en programas de aplicación. El significado es el que le quiera conferir el programador. | Fin del proceso que la recibe |
| *SIGSEGV* | 11 | **Fallo de segmentación**. Se envía a un proceso cuando éste intenta acceder a datos que están fuera de su segmento de datos. | Fin del proceso que la recibe y generación de fichero *core* |
| *SIGUSR2* | 12 | **Señal de usuario 2**. Es la segunda de las señales reservadas al usuario. Su acción y significado son similares a los de *SIGUSR1*. | Fin del proceso que la recibe |
| *SIGPIPE* | 13 | ***Pipe*** o ***socket*** **roto**. Se envía a un proceso cuando éste intenta escribir en un *pipe* del que no hay nadie leyendo. También se genera cuando un proceso intenta escribir en un *socket* del que no hay nadie leyendo. | Fin del proceso que la recibe |
| *SIGALRM* | 14 | **Alarma de reloj**. Se envía a un proceso cuando se sobrepasa el tiempo (en segundos) fijado mediante la función *alarm*. | Fin del proceso que la recibe |
| *SIGTERM* | 15 | **Terminar**. Se genera por defecto por la función *kill*. Es menos drástica que *SIGKILL* y puede ser ignorada. Se envía a todos los procesos durante la parada del sistema o *shutdown*. | Fin del proceso que la recibe |
| *SIGCHLD* | 17 | **Proceso hijo detenido** o **terminado**. Se genera cuando un proceso hijo ha parado de ejecutarse o ha terminado. | Ignorar la señal |

# 2.4.3. Señales

**¿Qué hace un proceso al recibir una señal?**

Si se envía señal a un proceso, si éste **no está preparado para aceptar dicha señal**, es la finalización de su ejecución → **proceso "muere"**.

Si el proceso dispone de un **gestor de señales**, el proceso responderá a la señal **ejecutando la rutina asignada** por el programador a las señales. Esta rutina podrá ser desarrollada por el propio programador, o bien será la asociada por defecto a la señal.

Después de capturar una señal se debe **restablecer el gestor de señales** mediante otra llamada a la función que asigna la señal al procedimiento. En caso contrario, se ejecutaría la acción por defecto de la señal.

# 2.4.3. Señales

## Función signal

Es un **gestor de señales.** Permite especificar la acción que debe realizarse cuando un proceso recibe una señal y lo prepara para recibir cierto tipo de señales.

Esto supone que será preciso añadir una llamada a **signal** para cada tipo de señal que se desee que el proceso reciba sin que cause su finalización.

```
#include <signal.h>
void (*signal (int signum, void (*func) (int))) (int);
```

Mediante **signal** se establece:

- Un gestor de señales para la señal **`signum`** (número de la señal).
- **`func`** permite elegir una de las tres acciones que se pueden realizar cuando se recibe la señal:
  **1) SIG_IGN**: ignora la señal
  **2) SIG_DFL**: devuelve a la señal su comportamiento por defecto
  **3) Función** especificada por usuario:

```
#include <signal.h>
void func (int sig);
```

# 3. Programación Concurrente

**Ejecución simultánea de múltiples tareas**.

Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa.

Las tareas se pueden ejecutar en una sola CPU (multiprogramación), en varios procesadores, o en una red de computadores distribuidos.

La **programación concurrente** es una forma de resolución de problemas lógicos, **ejecutando múltiples tareas a la misma vez y no de forma secuencial.**

En un programa concurrente las tareas puede continuar sin la necesidad que otras comiencen o finalicen**.**

# 3. Programación Concurrente

**BENEFICIOS**

- Mejor aprovechamiento de la CPU. Un proceso puede aprovechar ciclos de CPU mientras otro realiza una operación de entrada/salida.

- Velocidad de ejecución: al subdividir un programa en procesos, estos se pueden "repartir" entre procesadores o gestionar en un único procesador

- Solución a problemas de naturaleza concurrente
  - ✓ Sistemas de control: son sistemas en los que hay captura de datos, normalmente a través de sensores, análisis y actuación en función del análisis.

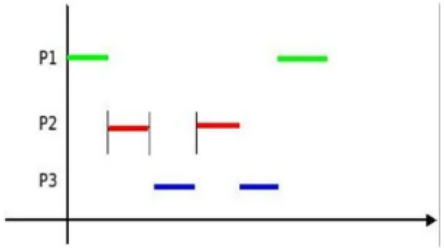
  - ✓ Tecnologías web: servidores web son capaces procesar múltiples peticiones de usuarios concurrentemente (servidores de chat, correo, navegadores web, …)

# 3. Programación Concurrente

## Concurrencia y hardware

Hardware y su topología es importante para abordar cualquier tipo de problema concurrente:

● **Sistemas monoprocesador**: podemos tener concurrencia, gestionando el tiempo de procesador para cada proceso




● Sistemas multiprocesador: un proceso en cada procesador. Éstos pueden ser de memoria compartida (fuertemente acoplados) o con memoria local a cada procesador (débilmente acoplados).

# 3. Programación Concurrente

**Problemas inherentes**

**Exclusión mutua**: en programación concurrente es muy típico que varios procesos accedan a la vez a una variable compartida para actualizarla.

Esto puede producir inconsistencia de datos: uno puede estar actualizando la variable a la vez que otro puede estar leyendo. Por ello es necesario **exclusión mutua** de los procesos respecto a la variable compartida. Solución: **región critica.**

Cuando dos o más procesos comparten una variable, el acceso a dicha variable debe efectuarse siempre dentro de la región critica asociada a la variable. Solo uno de los procesos podrá acceder para actualizarla y los demás deben esperar, el tiempo de estancia es finito.

# 3. Programación Concurrente

**Problemas inherentes**

**Condición de sincronización:** necesidad de coordinar los procesos con el fin de sincronizar sus actividades.

Puede ocurrir que un proceso P1 llegue a un estado X que no pueda continuar su ejecución hasta que otro proceso P2 haya llegado a un estado Y de su ejecución.

La programación concurrente proporciona mecanismos para bloquear procesos a la espera de que ocurra un evento y para desbloquearlos cuando este ocurra.

# 4. Programación Paralela y distribuida

Un **programa paralelo** es un **programa concurrente** diseñado para ejecutarse en un sistema multiprocesador. El procesamiento paralelo permite que muchos elementos de proceso independientes trabajen simultáneamente para resolver un problema

El problema a resolver se divide en partes independientes, de tal forma que cada elemento, pueda ejecutar la parte de programa que le corresponda a la vez que los demás.

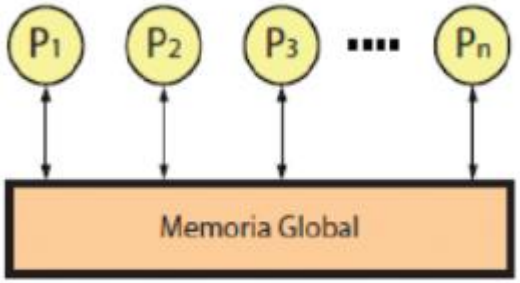
Cada procesador realiza una parte del problema y necesita intercambiar información con el resto. Diferentes **modelos de programación paralela**:

**Modelo de memoria compartida**: los procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones. Un valor escrito en memoria por un procesador puede ser leído directamente por cualquier otro.

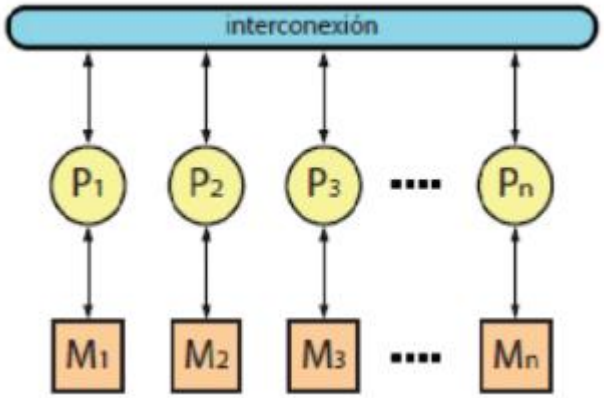
**Modelo de paso de mensajes**: cada procesador dispone de su propia memoria independiente del resto y accesible solo por el. Para realizar el intercambio de información es necesario que cada procesador realice la petición de datos al procesador que los tiene, y este haga el envío

# 4. Programación Paralela y distribuida

Procesadores comparten físicamente la memoria, es decir, todos acceden al mismo espacio de direcciones: **sistema de memoria compartida** o **multiprocesadores**.




Procesadores disponen de su propia memoria independiente del resto y accesible solo por el**: sistema de memoria distribuida** o **multicomputadoras**



# 4. Programación Paralela y distribuida

**Ventajas del procesamiento paralelo**:

- Proporciona ejecución simultanea de tareas.
- Disminuye el tiempo total de ejecución de una aplicación.
- Resolución de problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales, por ejemplo, los recursos que están en una red distribuida, una WAN o la propia red internet.
- Disminución de costos, en vez de gastar en un supercomputador muy caro se pueden utilizar otros recursos más baratos disponibles remotamente.

**Inconvenientes del procesamiento paralelo**:

- Los compiladores y entornos de programación para sistemas paralelos son mas difíciles de desarrollar.
- Los programas paralelos son más difíciles de escribir.
- Mayor complejidad en el acceso a los datos.
- La comunicación y la sincronización entre las diferentes subtareas.