# UT1. Programación Multiproceso

## Tabla de contenido

# 1. Introducción

Un programa contiene un conjunto de instrucciones que se pueden ejecutar directamente en una máquina. Es un objeto estático, normalmente almacenado en un fichero binario en un medio de almacenamiento secundario, como por ejemplo un disco duro.

Un proceso corresponde a una instancia de un programa en ejecución. La ejecución de un programa comienza con la creación y ejecución de un proceso y un proceso puede crear nuevos procesos sobre la marcha.

# 2. Programas. Ejecutables. Procesos y Servicios

Programa, proceso, ejecutable y servicio son términos que hacen referencia a elementos distintos, pero íntimamente relacionados. Es necesario comprender las particularidades de cada uno de ellos para poder avanzar en el conocimiento de la programación multiproceso.

El sistema operativo es el elemento del ordenador que coordina el funcionamiento del resto de componentes de este, tanto software como hardware. Es a él a quien se indica qué se quiere hacer o, siendo más precisos, qué programas se desean ejecutar.

Para poder ejecutar un programa primero hay que obtenerlo, o en el caso de los programadores, crearlo, para posteriormente proceder a su ejecución. El proceso de creación por parte del programador y de ejecución por parte del usuario final de un programa compilado es el siguiente:

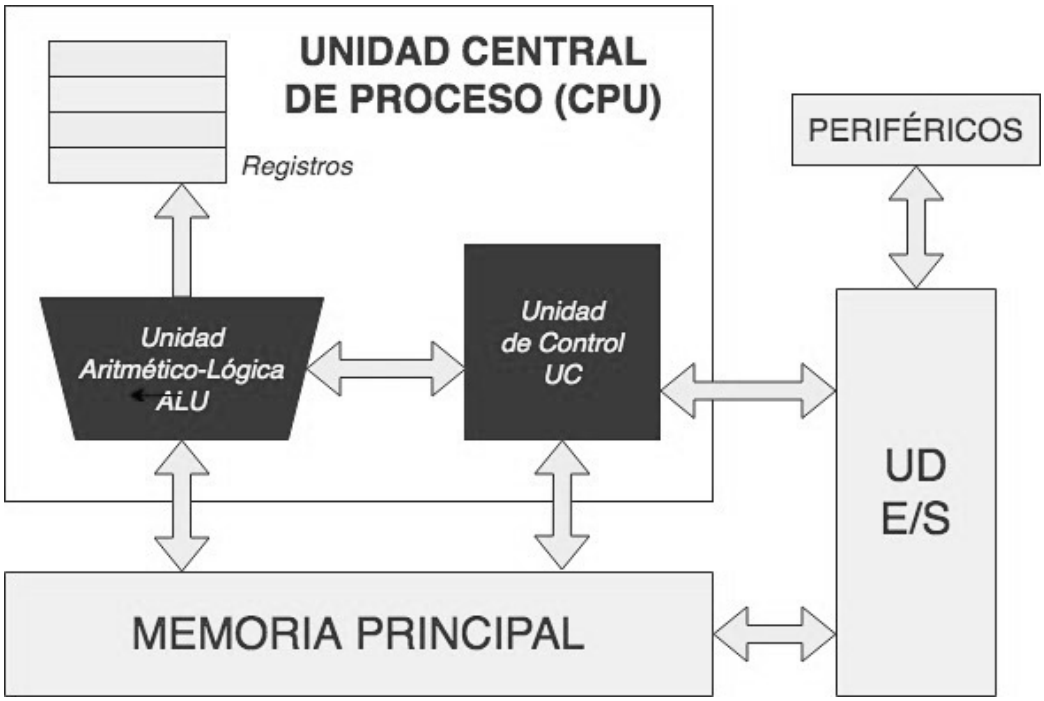
1. El programador escribe el código fuente con un editor de texto y lo almacena en un fichero.
2. El programador compila el código fuente utilizando un compilador, generando un programa ejecutable. Este programa contiene instrucciones comprensibles por el sistema operativo para el que se realizó la compilación.
3. El usuario ejecuta el programa ejecutable generando un proceso.

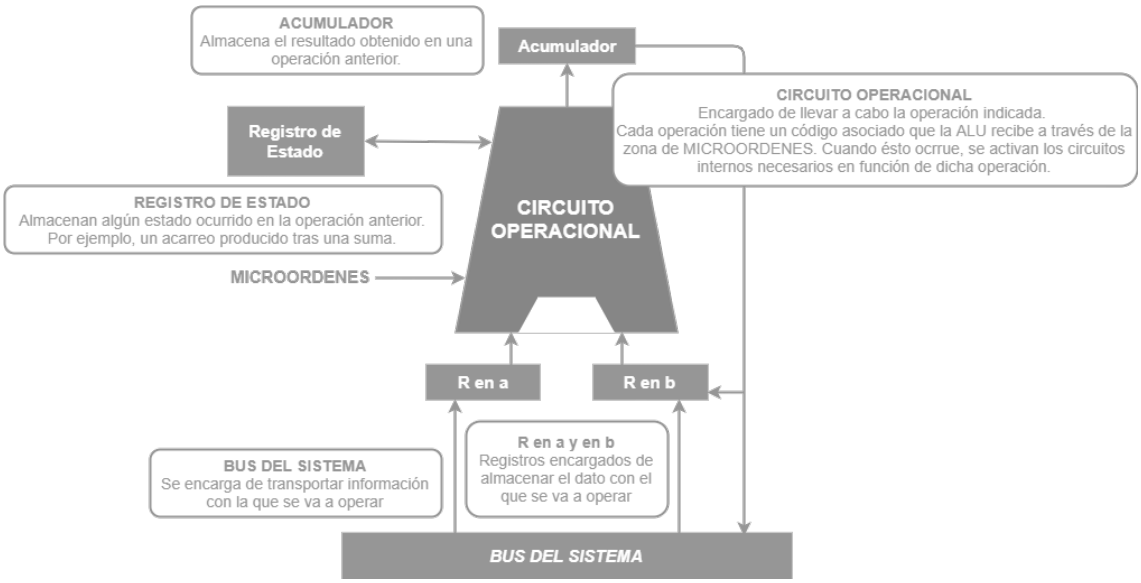
Por lo tanto, se puede afirmar que un programa, al ser ejecutado por un usuario, genera un proceso en el sistema operativo. Por su parte, un servicio es también un programa cuya ejecución se realiza en segundo plano y que no requiere interacción del usuario. Normalmente se arranca de manera automática por el sistema operativo y está en constante ejecución.

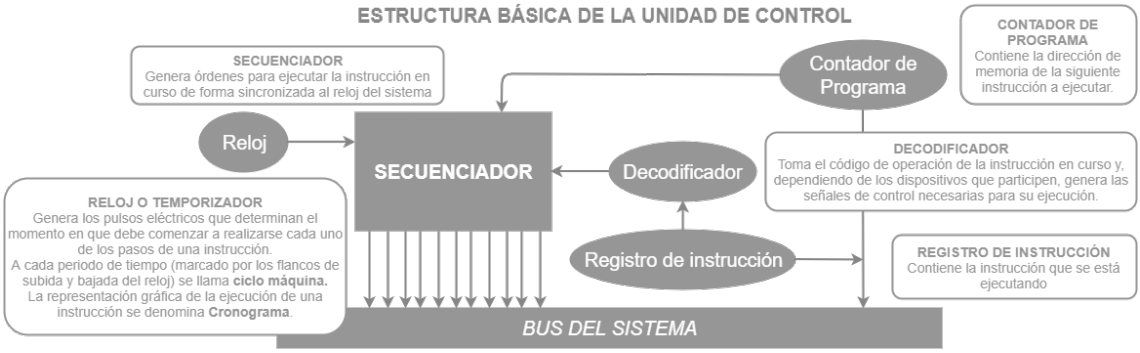
## 2.1 Ejecución de un programa

Para ejecutar un programa primero debe cargarse en memoria. Durante su ejecución utiliza diversos recursos del sistema:

- ***Memoria principal***. Consta de una secuencia de celdas de memoria, todas con la misma longitud en bits y cada una identificada por su posición. Antes de comenzar su ejecución, un programa debe cargarse en un bloque de la memoria principal, que se asigna al proceso que se crea para ejecutar el programa. Este proceso también puede obtener más memoria, dinámicamente durante su ejecución.
- ***Procesador o CPU (Central Processing Unit).*** Ejecuta el proceso que se crea una vez cargado el programa en la memoria. El procesador guarda en un registro especial, el contador de programa (PC o programam counter), la dirección en la memoria de la instrucción que se está ejecutando. U proceso no puede operar directamente con los contenidos de la memoria. Estos deben antes traerse de ella y cargarse en registros del procesador. Las operaciones se realizan en la unidad aritmético-lógica o ALU. Su resultado se obtiene en un registro desde el que se puede transferir a una posición de la memoria.
- ***Dispositivos de Entrada/Salida (ES).*** Los procesos comparten los dispositivos de E/S. Debe guardarse información acerca de a qué procesos se les ha otorgado acceso a un dispositivo de E/S y del estado de las operaciones realizadas sobre él.






## *1º FASE DE BÚSQUEDA DE INSTRUCCIÓN*

- Carga la dirección de la 1º instrucción en el contador de programa.
- El contenido del contador del programa pasa al registro de instrucciones (RI)
- Con ayuda de la ALU se incrementa el Contador de programa para apuntar a la siguiente instrucción.
- Se envía a memoria el contenido del Registro de instrucciones a través del bus de direcciones.
- Memoria devuelve la información requerida a través del bus de datos.
- La unidad de Control decodifica la instrucción y el secuenciador genera las órdenes para su ejecución.

## *2º FASE DE EJECUCIÓN DE LA INSTRUCCIÓN*

- Bajo la supervisión de la Unidad de Control, la ALU llevará a cabo la ejecución de las instrucciones. Esta ejecución varía dependiendo del código de operación.
- Cada vez que se realiza una operación, el registro de estado contendrá la información correspondiente que indica si se ha realizado o no, con éxito. Mientras el acumulador contendrá el resultado de la misma.

# 3. Programación concurrente, paralela y distribuida.

La ***MULTITAREA***, como indica su nombre, es la capacidad de realizar varias tareas simultáneamente, frente a la restricción de la monotarea, donde las tareas se ejecutan una detrás de otra. Cuando se trabaja en un sistema multitarea, varias tareas avanzan a la vez aunque pueden hacerlo de diferentes maneras.

## 3.1 Concurrencia

Un sistema basado en un único procesador, con un único núcleo es capaz de realizar multitarea mediante el uso de la ***CONCURRENCIA***. En la computación concurrente, los tiempos de CPU se reparten entre los distintos procesos según una planificación dirigida por el sistema operativo. En una unidad de tiempo de computación solo avanza un proceso. La velocidad en la asignación de la CPU a los distintos procesos logra queno se perciba el cambio.

### PROBLEMAS DE LA PROGRAMACIÓN CONCURRENTE

- ***Exclusión Mutua***: es el resultado de que dos procesos intenten acceder a la misma variable. Esto puede producir inconsistencia puesto que un proceso puede estar actualizando una variable mientras que otro proceso está leyéndola. Para evitar este problema se utiliza la **región crítica**, que controla el número de procesos que están utilizando el recurso.
- ***Abrazo Mortal***: dos procesos se quedan bloqueados porque están esperando a los recursos que tiene el otro proceso. También es denominado DeadLock o interbloqueo.
- ***Inanición***: un proceso se queda esperando un recurso compartido que siempre se le deniega. Sin este recurso el proceso no puede finalizar.

## 3.2 Procesamiento en paralelo

Los sistemas basados en varios procesadores o en procesadores de varios núcleos aportan una mejora sustancial: permiten ejecutar varias instrucciones en un único ciclo de reloj. Esta capacidad hace posible ejecutar en ***paralelo*** varias instrucciones. En este tipo de procesamiento, los procesos se dividen en pequeñas subtareas (***hilos***) que se ejecutan en los diferentes núcleos, consiguiendo una reducción en los tiempos de ejecución de los procesos.

### MODELOS DE INTERCAMBIO DE INFORMACIÓN

- ***Modelo de memoria compartida***: este tipo de memoria permite ser accedida por múltiples programas, por lo que es un modo eficaz de transferencia de datos. Esto se consigue creando un espacio de acceso común por parte de cada uno de loa procesos en la memoria RAM.
  Los puntos de acceso común son las **SECCIONES CRÍTICAS**. Es necesario establecer controles dentro de un punto de acceso a datos común, puesto que los procesos intentarán realizar lectura y escritura de datos a la vez, dando lugar a datos incoherentes. Existen diversos mecanismos de control: **semáforos**, tuberías, monitores, etc. Estos permitirán el acceso de los hilos de manera individual, asegurando que las operaciones realizadas son **atómicas**, es decir, hasta que un hilo no termina, otro no puede acceder al mismo espacio de memoria.
- ***Modelo de paso de mensaje:*** es el mecanismo más utilizado en la programación orientada a objetos. El paso de mensajes implica dos operaciones: enviar y recibir mensajes. Un mensaje es una unidad de datos que contiene información relevante para la comunicación. Puede ser de tamaño fijo o variable, y puede tener un formato o estructura específica. Un proceso puede enviar un mensaje a otro proceso especificando su identificador o a un grupo de procesos mediante un mecanismo de difusión o multidifusión. Un proceso puede recibir un mensaje de otro proceso especificando su identificador o de cualquier proceso mediante un comodín o un filtro. El paso de mensajes puede ser sincrónico o asincrónico, dependiendo de si el remitente y el receptor bloquean o continúan su ejecución después de la operación.

### VENTAJAS DE LA PROGRAMACIÓN PARALELA

- Permite la ejecución de tareas de manera simultánea.
- Permite resolver problemas complejos.
- Disminuye el tiempo en ejecución.

### INCONVENIENTES DE LA PROGRAMACIÓN PARALELA

- Mayor dificultad en la programación.
- Mayor complejidad en el acceso a datos.

## 3.3 Programación distribuida

La ***PROGRAMACIÓN DISTRIBUIDA*** es otro de los paradigmas de la programación multiproceso. En este tipo de arquitectura, la ejecución del software se distribuye entre varios ordenadores, consiguiendo así disponer de una potencia de procesamiento mucho más elevada, escalable y, normalmente económica. Para construir un sistema distribuido se requiere una red de ordenadores entre los que distribuir el trabajo.

### CARACTERÍSTICAS DE LOS SISTEMAS DISTRIBUIDOS

- ***Capacidad de balanceo***: las propias máquinas son capaces de realizar una asignación de recursos concreta para cada proceso, de manera que otro proceso pueda hacer uso de una mayor cantidad de recursos si lo requiere.
- ***Alta disponibilidad***: es una de las características más usadas, que permite a este tipo de programación una flexibilidad enorme, en caso de que haya un fallo, automáticamente los procesos son asignados en otros servidores.

### VENTAJAS DE LA PROGRAMACIÓN DISTRIBUIDA

- Permite escalabilidad, capacidad de crecimiento.
- Permite la compartición de recursos y datos.
- Mayor flexibilidad.
- Alta disponibilidad.

### INCONVENIENTES DE LA PROGRAMACIÓN DISTRIBUIDA

- Pérdida de mensajes.
- Está expuesta a diferentes ataques para vulnerar su seguridad.

# 4. Procesos y Sistema Operativo

Cuando se suspende temporalmente la ejecución de un proceso, éste debe rearrancarse en el mismo estado en que se encontraba cuando se paró, esto implica que toda la información referente al proceso debe almacenarse en alguna parte.

El **BCP** es una estructura de datos llamada ***BLOQUE DE CONTROL DE PROCESO*** donde se almacena información acerca de un proceso:

- Identificación del proceso.
- Estado del proceso.
- Contador de programa.
- Registros de CPU.
- Información de planificación como la prioridad.
- Información de gestión de memoria.
- Información contable como la cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

Mediante el comando **ps (process status)** de Linux podemos ver parte de la información asociada a cada proceso. Añadiendo las opciones AF (**ps -af**) se muestran todos los detalles.

- **PID**: identificador del proceso.
- **TTY**: terminal asociado del que lee y al que escribe. Si no hay aparece interrogación.
- **TIME**: tiempo de ejecución asociado, es la cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació.
- **CMD**: nombre del proceso.
- **UID**: nombre de usuario.
- **PPID**: PID del padre de cada proceso.
- **C:** porcentaje de recursos CPU utilizado por el proceso.
- **STIME**: hora de inicio del proceso.
- **SZ**: tamaño virtual de la imagen del proceso.
- **RSS**: tamaño de la parte residente en memoria en kilobytes.
- **PSR**: procesador que el proceso tiene actualmente asignado.

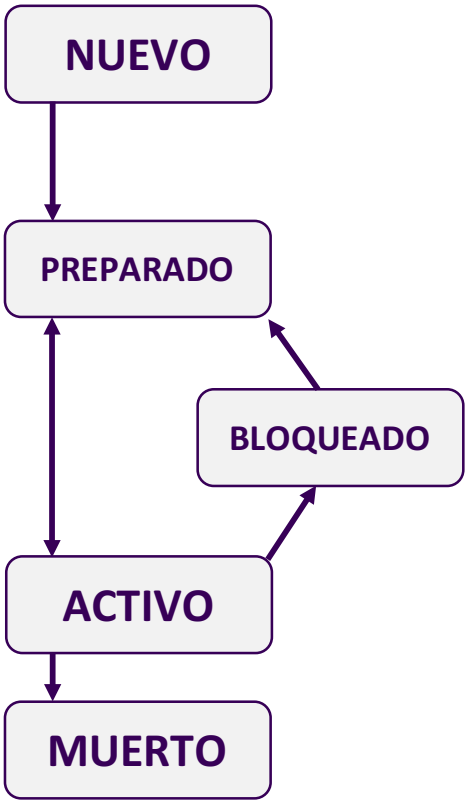
En **Ubuntu** podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos desde **Sistema > Administración > Monitor del sistema**.

En sistemas operativos Windows podemos usar desde la línea de comandos la orden **tasklist** para ver los procesos que se están ejecutando. Aunque lo más típico es utilizar la combinación de teclas **[CTRL + Alt + Supr]** para que se muestre la pantalla que da acceso al **Administrador de tareas de Windows**.

## 4.1 Identificación de los estados de un proceso

El Sistema Operativo se conoce como SOFTWARE BASE. Es un conjunto de programas que se inician al arrancar el ordenador y cuya misión es desvincular al usuario las características hardware del equipo y facilitar la ejecución de los programas. Sus principales funciones son:

- **Gestión de procesos**
- Gestión de memoria
- Gestión de Entrada/Salida
- Gestión de archivos
- Gestión de seguridad




- Cuando se crea un proceso es ***nuevo*** y una vez esté listo para su ejecución (cargado en memoria principal) se dice que está ***preparado*** (espera para ejecutarse)
- Estará ***activo*** cuando pasa a ocupar la CPU y se está ejecutando.
- Si pasa de activo a ***bloqueado***, estará esperando alguna operación de E/S. Cuando pueda volver a ejecutarse pasará a esperar su turno en estado preparado.
- Si pasa de activo a preparado el proceso no necesita nada, pero el sistema operativo es de tiempo compartido y reparte la CPU entre todos los procesos.
- Si finaliza su ejecución (correcta o incorrectamente) pasa a estar ***muerto***.

## 4.2 Algoritmos de gestión de procesos

- A la forma en que la CPU se distribuye para ejecutar los procesos se llama ***planificación***.
- El planificador a corto plazo (***dispatcher***) decide qué proceso de los que están preparados para ejecutarse pasa a utilizar el procesador. Éste debe intentar minimizar el tiempo de respuesta, maximizar la cantidad de procesos que se ejecutan y evitar que un proceso quede postegrado indefinidamente.
- El ***envejecimiento*** es una técnica en la que un proceso con poca prioridad, a medida que espera más tiempo para entrar a ocupar el procesador, va ganando prioridad, para así evitar que quede esperando indefinidamente por 'colarse' procesos más prioritarios.

Se conoce como **ALGORITMO DE PLANIFICACIÓN DE PROCESOS** al conjunto de reglas que indican qué proceso debe entrar a ejecutarse de entre los que se encuentran en espera. Las planificaciones de procesos se clasifican en **EXPULSIVAS (PREENPTIVE**) cuando un proceso puede ser desalojado de la CPU sin haber finalizado y **NO EXPULSIVAS (NON PREENPTIVE)** cuando no puede ser desalojado hasta que finalice su ejecución.

**SE DISTINGUE ENTRE:**

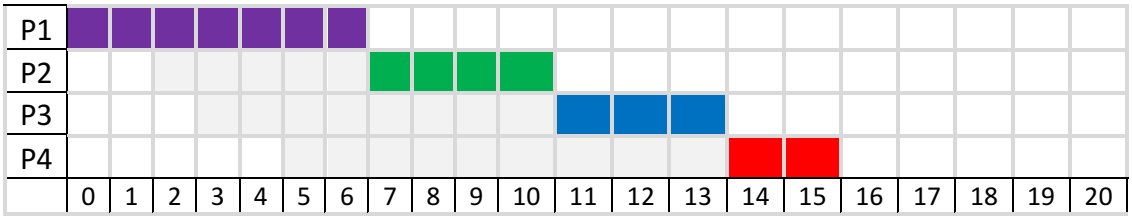
- **MOMENTO DE LLEGADA (ti):** tiempo en que el proceso pasa a estar listo para su ejecución.
- **MOMENTO DE FINALIZACIÓN (tf):** tiempo en que el proceso acaba su ejecución y pasa a estado muerto.
- **TIEMPO DE EJECUCIÓN (t):** cantidad de tiempo que el proceso necesita para su ejecución completa.
- **MOMENTO DE SERVICIO (T=tf-ti):** Tiempo ocurrido desde el momento en que el proceso está listo hasta que finaliza su ejecución.
- **TIEMPO DE ESPERA (E=T-t):** Tiempo transcurrido entre que el proceso está listo y finaliza su ejecución y NO ha estado en la CPU.
- **ÍNDICE DE SERVICIO (I=t/T):** Cociente entre el tiempo de ejecución y de servicio. Mide la efectividad del algoritmo.

## ALGORITMOS DE PLANIFICACIÓN:

### FCFS: PRIMERO EN ENTRAR, PRIMERO EN SALIR.

Los procesos se ejecutan en orden de llegada. Es no expulsivo; es decir, es APROPIATIVO.

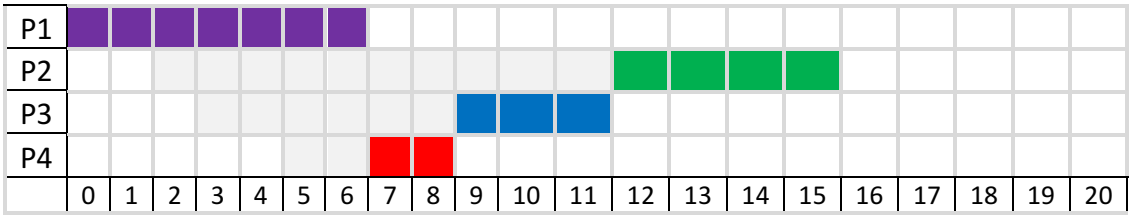
| PROCESO | ti | t | tf | T | E | I |
|---|---|---|---|---|---|---|
| P1 | 0 | 7 | 7 | 7-0 = 7 | 7-7 = 0 | 7/7 |
| P2 | 2 | 4 | 11 | 11-2 = 9 | 9-4 = 5 | 4/9 |
| P3 | 3 | 3 | 14 | 14-3 = 11 | 11-3 = 8 | 3/11 |
| P4 | 5 | 2 | 16 | 16-5 = 11 | 11-2 = 9 | 2/11 |




### SJF: PRIMERO EL PROCESO MÁS CORTO.

En caso de igual tiempo se aplica FCFS. Es NO expulsivo.

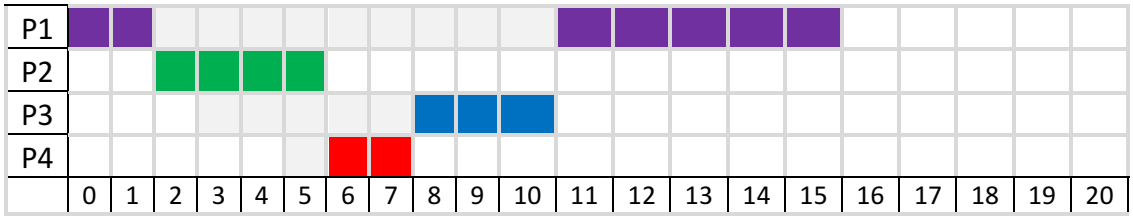
| PROCESO | ti | t | tf | T | E | I |
|---|---|---|---|---|---|---|
| P1 | 0 | 7 | 7 | 7-0 = 7 | 7-7 = 0 | 7/7 |
| P2 | 2 | 4 | 16 | 16-2 = 14 | 14-4 = 10 | 4/14 |
| P3 | 3 | 3 | 12 | 12-3 = 9 | 9-3 = 6 | 3/9 |
| P4 | 5 | 2 | 9 | 9-5 = 4 | 4-2 = 2 | 2/4 |



## SRTF: PRIMERO EL TIEMPO RESTANTE MÁS CORTO.

En caso de empate se aplica FCFS. Es expulsivo.

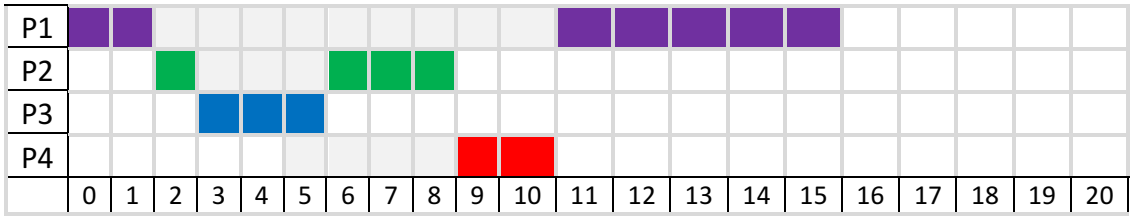
| PROCESO | ti | t | tf | T | E | I |
|---|---|---|---|---|---|---|
| P1 | 0 | 7 | 16 | 16-0 = 16 | 16-7 = 9 | 7/16 |
| P2 | 2 | 4 | 6 | 6-2 = 4 | 4-4 = 0 | 4/4 |
| P3 | 3 | 3 | 11 | 11-3 = 8 | 8-3 = 5 | 3/8 |
| P4 | 5 | 2 | 8 | 8-5 = 3 | 3-2 = 1 | 2/3 |




## POR PRIORIDADES expulsivo:

El orden de entrada es según prioridad ye n caso de empate se aplica FCFS. Puede ser o no expulsivo y variar las prioridades de los procesos a lo largo de la ejecución del algoritmo.

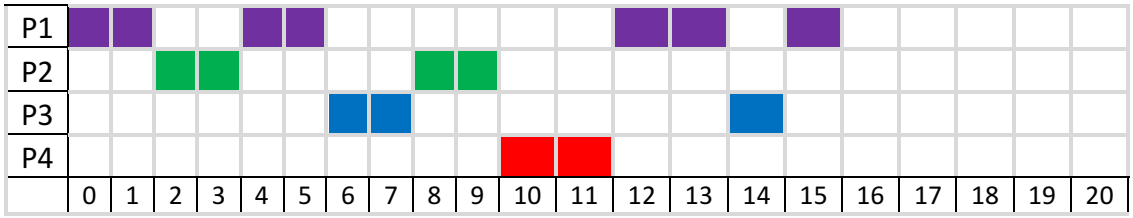
| PROCESO | ti | t | prioridad | tf | T | E | I |
|---|---|---|---|---|---|---|---|
| P1 | 0 | 7 | 4 | 16 | 16-0 = 16 | 16-7 = 9 | 7/16 |
| P2 | 2 | 4 | 2 | 9 | 9-2 = 7 | 7-4 = 3 | 4/7 |
| P3 | 3 | 3 | 1 | 6 | 6-3 = 3 | 3-3 = 0 | 3/3 |
| P4 | 5 | 2 | 3 | 11 | 11-5 = 6 | 6-2 = 4 | 2/6 |




## ROUND ROBIN. POR RONDAS:

Va repartiendo a los procesos el tiempo de CPU a partes iguales. Cada parte de tiempo se llama quantum y es indivisible. Es expulsivo.

| PROCESO | ti | t | tf | T | E | I |
|---|---|---|---|---|---|---|
| P1 | 0 | 7 | 16 | 16-0 = 16 | 16-7 = 9 | 7/16 |
| P2 | 2 | 4 | 10 | 10-2 = 8 | 8-4 = 4 | 4/8 |
| P3 | 3 | 3 | 15 | 15-3 = 12 | 12-3 = 9 | 3/12 |
| P4 | 5 | 2 | 12 | 12-5 = 7 | 7-2 = 5 | 2/7 |



# 5. PROGRAMACIÓN DE PROCESOS EN C

## ACTIVIDAD GUIADA 1.

La misión de la función **fork()** es crear un proceso nuevo. Su sintaxis es

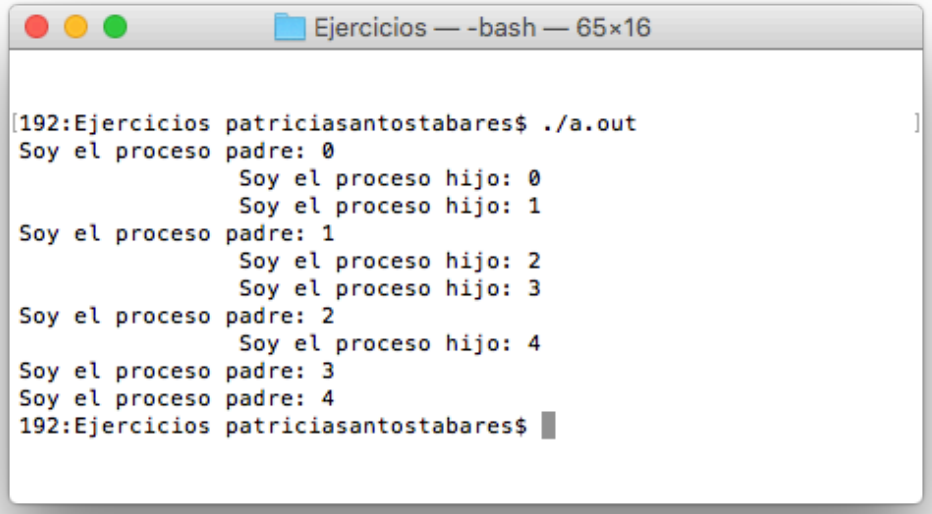
```
#include <unistd.h>
pid_t fork(void);
```

Al llamar a esta función se crea un nuevo proceso (proceso hijo) que es una copia exacta en código y datos del proceso que ha realizado la llamada (el proceso padre), salvo el PID y la memoria que ocupa. Las variables del proceso hijo son una copia de las del padre, por lo que modificar una variable en uno de los procesos no se refleja en el otro.

El valor devuelto por fork() es un valor numérico:

- Devuelve -1 si se produce algún error.
- Devuelve 0 si no se produce ningún error y nos encontramos en el proceso hijo.
- Devuelve el PID asignado al proceso hijo si no se produce ningún error y nos encontramos en el proceso padre.

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

int main()
{
    int i=0;

    switch(fork()) {
        case -1:
                perror("Error al crear el proceso\n");
                exit(-1);
                break;
        case 0:  /* Codigo para el hijo */
                while(i<5){
                    sleep(1);
                    printf("\t\tSoy el proceso hijo: %d\n", i++);
                }
                break;
        default: /* Codigo para el padre */
                while(i<5){
                   printf("Soy el proceso padre: %d\n", i++);
                   sleep(2);
                }
    };
    exit(0);
}
```




## ACTIVIDAD GUIADA 2.

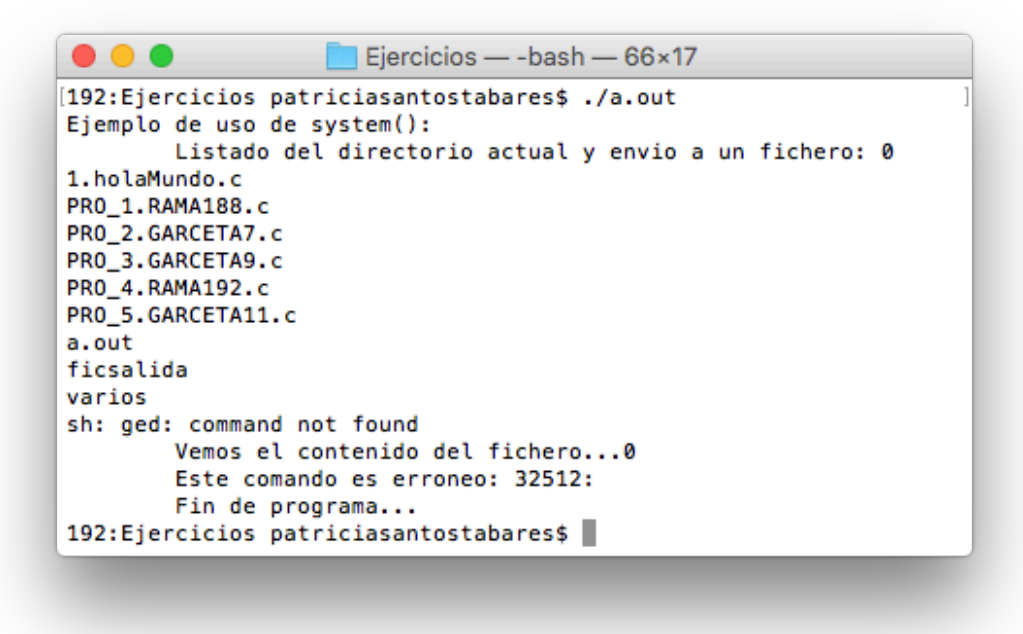
Seguro que más de una vez hemos necesitado dentro de un programa ejecutar otro programa que realice alguna tarea concreta. Linux ofrece varias funciones para realizar esto: **system(), fork(), execl().**

La función system() se encuentra en la librería estándar **stdlib.h** por lo que funciona en cualquier sistema operativo que tenga un compilador de C/C++ como por ejemplo Linux, Windows, etc. Su prototipo es: **int system (const char *cadena).**

La función recibe como parámetro una cadena de caracteres que indica el comando que se desea procesar. Dicha instrucción es pasada al intérprete de comandos del ambiente en el que se esté trabajando y se ejecuta. Devuelve el valor -1 si ocurre un error y el estado devuelto por el comando en caso contrario.

La ejecución del siguiente ejemplo en C lista el contenido del directorio actual y lo envía a un fichero, abre el editor gedit con el fichero generado y ejecuta un comando que no existe en el intérprete de comandos de Linux.

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

int main()
{
    printf("Ejemplo de uso de system(): ");
    printf("\n\tListado del directorio actual y envio a un fichero: ");
    printf("%d", system("ls>ficsalida"));
    printf("\n\tVemos el contenido del fichero...");
    printf("%d", system("cat ficsalida"));
    printf("\n\tEste comando es erroneo: %d:", system("ged"));
    printf("\n\tFin de programa...\n");
}
```



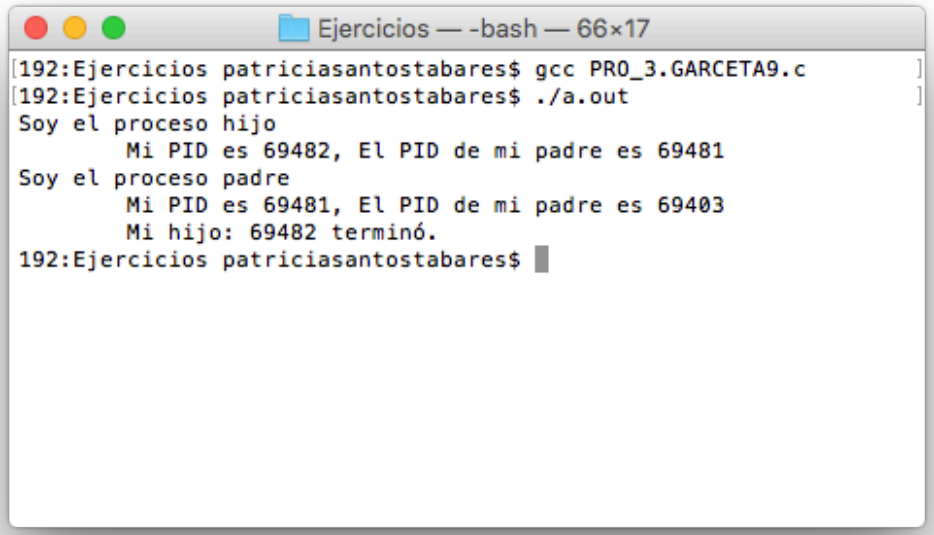

Esta función no se debe usar desde un programa con privilegios de administrador porque pudiera ser que se emplearan valores extraños para algunas variables de entorno y podrían comprometer la integridad del sistema. En este caso se utiliza **execl().**

## ACTIVIDAD GUIADA 3.

Al realizar este nuevo ejemplo con la función fork() vamos a ver cómo obtener el identificador de un proceso o PID. Para ello utilizamos dos funciones que devuelven un tipo pid_t que son:

**`pid_t getpid(void);`**
Devuelve el identificador del proceso que realiza la llamada, es decir del proceso actual.
**`pid_t getppid(void);`**
Devuelve el identificador del proceso padre del proceso actual.

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

int main()
{
    pid_t pid, hijo_pid;
    pid = fork();

    if(pid==-1) {
        //Ha ocurrido un error
        printf("No se ha podido crear el proceso hijo...\n");
        exit(-1);
    }
    else if(pid==0) {
        //Soy el proceso hijo
        printf("Soy el proceso hijo\n\tMi PID es %d, El PID de mi padre es %d\n",
        getpid(), getppid());
    }
    else {
        //Nos encontramos en proceso padre
        hijo_pid=wait(NULL);//Espera la finalizacion del proceso hijo
        printf("Soy el proceso padre\n\tMi PID es %d, El PID de mi padre es
        %d\n\tMi hijo: %d terminó.\n", getpid(), getppid(), pid);
        exit(0);
    }
}
```

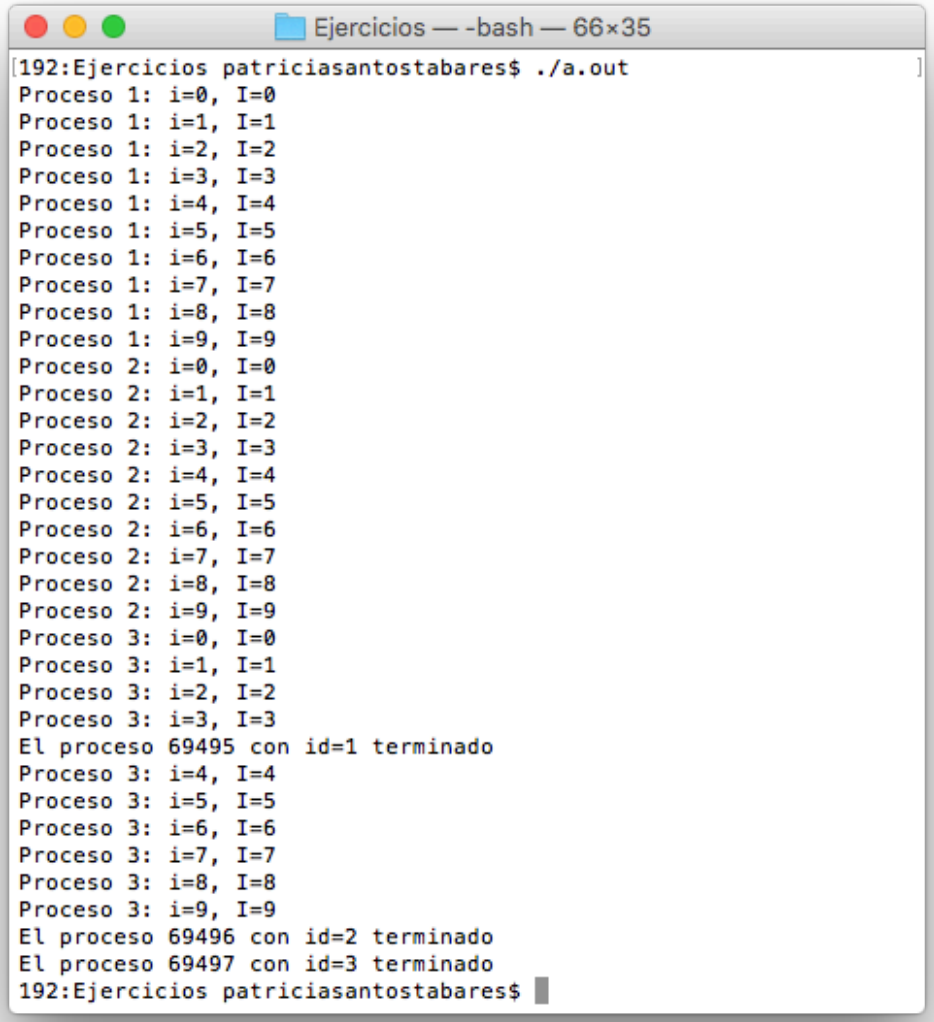



En el código anterior se utiliza la función **wait()** para que el proceso padre espere la finalización del proceso hijo, el proceso padre quedará bloqueado hasta que termine el hijo. La sintaxis de la orden es la siguiente: **pid_t wait (int *status);**

Devuelve el identificador del proceso hijo cuya ejecución ha finalizado. La sentencia wait(NULL) es la forma más básica de esperar que un hijo termine.

# ACTIVIDAD GUIADA 4.

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

#define NUM_PROCESOS 3
int I=0;

void codigo_del_proceso(int id)
{
    int i;
    for(i=0; i<10; i++)
        printf("Proceso %d: i=%d, I=%d\n", id, i, I++);
    exit(id);
}

int main()
{
    int p;
    int id[NUM_PROCESOS]={1,2,3};
    int pid;
    int salida;

    for(p=0; p<NUM_PROCESOS; p++) {
        pid=fork();
        if(pid==-1) {
            perror("Error al crear el proceso");
            exit(-1);
        } else if(pid==0) {
            /*Codigo del proceso hijo*/
            codigo_del_proceso(id[p]);
        }
    }
    /*Esta parte solo la ejecuta el proceso padre*/
    for(p=0; p<NUM_PROCESOS; p++) {
        pid=wait(&salida);
        printf("El proceso %d con id=%x terminado\n", pid, salida>>8);
    }
}
```


Ejercicios — -bash — 66×35
192:Ejercicios patriciasantostabares$ ./a.out
Proceso 1: i=0, I=0
Proceso 1: i=1, I=1
Proceso 1: i=2, I=2
Proceso 1: i=3, I=3
Proceso 1: i=4, I=4
Proceso 1: i=5, I=5
Proceso 1: i=6, I=6
Proceso 1: i=7, I=7
Proceso 1: i=8, I=8
Proceso 1: i=9, I=9
Proceso 2: i=0, I=0
Proceso 2: i=1, I=1
Proceso 2: i=2, I=2
Proceso 2: i=3, I=3
Proceso 2: i=4, I=4
Proceso 2: i=5, I=5
Proceso 2: i=6, I=6
Proceso 2: i=7, I=7
Proceso 2: i=8, I=8
Proceso 2: i=9, I=9
Proceso 3: i=0, I=0
Proceso 3: i=1, I=1
Proceso 3: i=2, I=2
Proceso 3: i=3, I=3
El proceso 69495 con id=1 terminado
Proceso 3: i=4, I=4
Proceso 3: i=5, I=5
Proceso 3: i=6, I=6
Proceso 3: i=7, I=7
Proceso 3: i=8, I=8
Proceso 3: i=9, I=9
El proceso 69496 con id=2 terminado
El proceso 69497 con id=3 terminado
192:Ejercicios patriciasantostabares$

# ACTIVIDAD GUIADA 5.

Partiendo de la Actividad guiada 3, creamos un nuevo proceso en el proceso hijo para tener el proceso padre (ABUELO), el proceso hijo (HIJO) y el proceso hijo del hijo (NIETO)

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

//ABUELO-HIJO-NIETO
int main()
{
    pid_t pid, hijoPID, pid2, hijo2PID;

    pid=fork();//Soy el abuelo, creo al hijo

    if(pid==-1) {
        printf("No se ha podido crear el proceso hijo...");
        exit(-1);
    }
    if(pid==0) {
        //Nos encontramos en el proceso hijo
        pid2=fork();//Soy el abuelo, creo al hijo

        if(pid2==-1) {
            printf("No se ha podido crear el proceso hijo...");
            exit(-1);
        }
        if(pid2==0) {
            //Nos encontramos en el proceso hijo del hijo(NIETO)
            printf("Soy el proceso NIETO %d; mi padre es=%d; mi ID de grupo es:%d\n",
            getpid(), getppid(), getpgrp());
        }
        else {
            //Nos encontramos en el proceso padre
            hijo2PID=wait(NULL); //Espera la finalización de proceso hijo
            printf("Soy el proceso PADRE: %d. mi padre es=%d; mi ID de grupo es:%d. Mi HIJO:%d Termino.\n",
            getpid(), getppid(), getpgrp(), pid2);
        }
    }
    else {
        //Nos encontramos en el proceso padre (ABUELO)
        hijoPID=wait(NULL); //Espera la finalización de proceso hijo
        printf("Soy el proceso ABUELO: %d. mi padre es=%d; mi ID de grupo es:%d. Mi HIJO:%d Termino.\n",
        getpid(), getppid(), getpgrp(), pid);
    }
}
```

```
Ejercicios — -bash — 97×7
192:Ejercicios patriciasantostabares$ ./a.out
Soy el proceso NIETO 69507; mi padre es=69506; mi ID de grupo es:69505
Soy el proceso PADRE: 69506. mi padre es=69505; mi ID de grupo es:69505. Mi HIJO:69507 Termino.
Soy el proceso ABUELO: 69505. mi padre es=69403; mi ID de grupo es:69505. Mi HIJO:69506 Termino.
192:Ejercicios patriciasantostabares$
```

## ACTIVIDAD PROPUESTA

Realiza un programa en C que cree un proceso (tendremos 2 procesos, uno padre y uno hijo). El programa definirá una variable entera y le dará el valor 6. El proceso padre incrementará dicho valor en 5 y el hijo restará 5. Se deben mostrar los valores en pantalla.

# 6. COMUNICACIÓN ENTRE PROCESOS

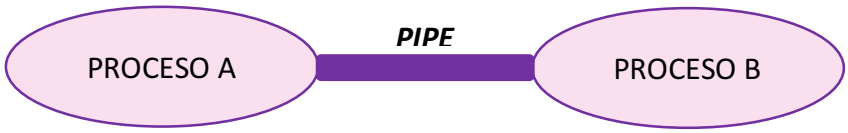
Por definición, los procesos de un sistema son elementos estancos. Cada uno tiene su espacio de memoria, su tiempo de CPU asignado por el planificador y su estado de los registros. No obstante, los procesos deben poder comunicarse entre sí, ya que es natural que surjan dependencias entre ellos en lo referente a las entradas y salidas de datos.

La comunicación entre procesos se denomina IPC (Inter-Process Communication) y existen diversas alternativas para llevarla a cabo. Algunas de estas alternativas son las siguientes:

- **Utilización de sockets**. Los sockets son mecanismos de comunicación de bajo nivel. Permiten establecer canales de comunicación de bytes bidireccionales entre procesos alojados en distintas máquinas programados con diferentes lenguajes.
- **Utilización de flujos de entrada y salida**. Los procesos pueden interceptar los flujos de entrada y salida estándar, por lo que pueden leer y escribir información unos en otros. En este caso, los procesos deben estar relacionados previamente (uno debe haber arrancado al otro obteniendo referencia al mismo).
- **RPC. (Remote Process Call).** Llamada a procedimiento remoto. Consiste en realizar llamadas a métodos de otros procesos que, potencialmente pueden estar ejecutándose en otras máquinas. En Java, este tipo de llamada se realiza mediante tecnología conocida como RMI (Remote Method Invocation), equivalente a RPC pero orientada a objetos.
- **Mediante el uso de sistemas de persistencia**. Consiste en realizar escrituras y lecturas desde los distintos procesos en cualquier tipo de sistema de persistencia, como ficheros o bases de datos. Pese a su sencillez, no se puede ignorar esta alternativa, ya que puede ser suficiente en múltiples ocasiones.
- **Mediante el uso de servicios proporcionados a través de internet.** Los procesos pueden utilizar servicios de transferencia de ficheros FTP, aplicaciones o servicios web, así como la tecnología cloud como mecanismos de conexión entre procesos que permiten el intercambio de información.

## 6.1 Tuberías o *pipes*

Un pipe es una especie de falso fichero que sirve para conectar dos procesos. Si el proceso A quiere enviar datos al proceso B, los escribe en el pipe como si este fuera un fichero de salida. El proceso B puede leer los datos sin más que leer el pipe como si se tratara de un fichero de entrada. Así, la comunicación entre procesos es parecida a la lectura/escritura en ficheros normales.




Cuando un proceso quiere leer del pipe y este está vacío, tendrá que esperar (es decir, se bloqueará) hasta que algún otro proceso ponga datos en él. Igualmente, cuando un proceso intenta escribir en el pipe y está lleno, se bloqueará hasta que se vacíe. El pipe es bidireccional, pero cada proceso lo utiliza en una única dirección.

```
#include <unistd.h>
int pipe (int fd[2]);
```

Esta función recibe un solo argumento, que es un array de dos enteros: fd[0] contiene el descriptor para lectura y fd[1] el de escritura. Si la función tiene éxito devuelve 0 y el array contendrá dos nuevos descriptores de archivos para ser usados por la tubería. Si ocurre algún error devuelve -1.

Para enviar datos al pipe, se usa la función write() y para recuperar datos del pipe, se usa la función read(). Su sintaxis es la siguiente:

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

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

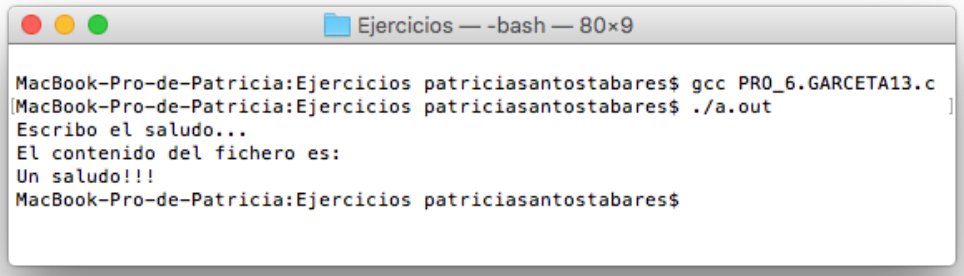
**write()** es muy similar. A buf le damos el valor de lo que queramos escribir, definimos su tamaño en count y especificamos el fichero en el que escribiremos en fd.

Antes de ver un ejemplo, se muestran las funciones que abren y cierran ficheros:

```
int open (const char *fichero, int modo);
int close (int fd);
```

**open()** abre el fichero indicado en la cadena fichero segúne l modo de acceso indicando en el entero modo (0 para lectura, 1 para escritura, 2 para lectura y escritura) Devuelve -1 si ocurre algún error. Para cerrar el fichero usamos close() indicando entre paréntesis el descriptor del fichero a cerrar.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    char saludo[]="Un saludo!!!\n";
    char buffer[10];
    int fd, bytesleidos;

    fd=open("texto.txt", 1); //El fichero se abre en modo escritura
    if(fd==-1){
        printf("ERROR al abrir el fichero...\n");
        exit(-1);
    }

    printf("Escribo el saludo...\n");
    write(fd, saludo, strlen(saludo));
    close(fd); //Cierro el fichero

    fd=open("texto.txt", 0); //El fichero se abre en modo lectura
    printf("El contenido del fichero es: \n");
    //leo bytes de uno en uno y lo guardo en el buffer
    bytesleidos=read(fd, buffer, 1);
    while(bytesleidos){
        printf("%s", buffer); //escribo el byte leido
        bytesleidos=read(fd, buffer, 1); //leo otro byte
    }
    close(fd);
}
```



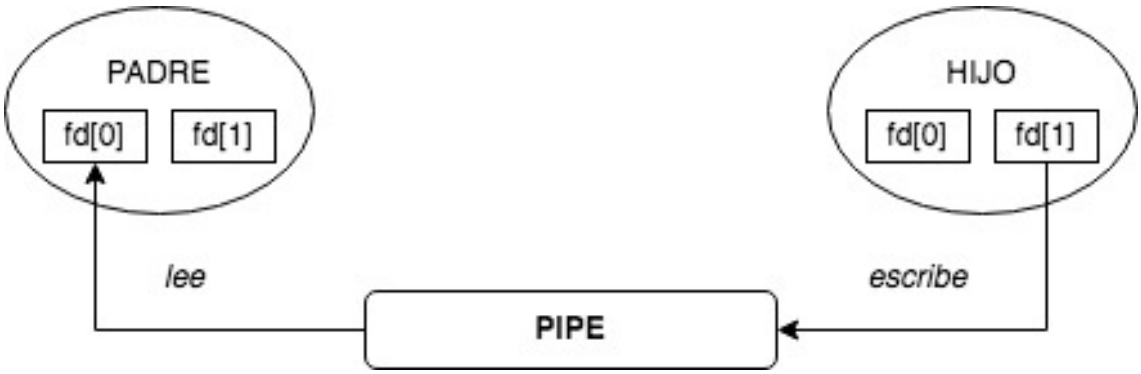
## ACTIVIDAD GUIADA 6.

Una vez sabemos cómo leer y escribir en ficheros, veamos algunos ejemplos usando pipes. En el primer ejemplo se crea un proceso hijo con fork(). El proceso hijo envía al proceso padre, mediante el uso de pipes, el mensaje “Hola papi” en el descriptor para escritura fd[1], el proceso padre, mediante el descriptor fd[0] lee los datos enviados por el hijo:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fd[2];
    char buffer[30];
    pid_t pid;
    pipe(fd); //Se crea el pipe
    pid=fork(); //Se crea el proceso hijo

    switch(pid)
    {
        case -1: //ERROR
                printf("No se ha podido crear el hijo...");
                exit(-1);
                break;
        case 0: //HIJO
                printf("El hijo escribe en el pipe...\n");
                write(fd[1], "Hola papi", 10);
                break;
        default: //PADRE
                wait(NULL); //Espera que finalice el proceso hijo
                printf("El padre lee del pie...\n");
                read(fd[0], buffer, 10);
                printf("El mensaje leido es: %s\n", buffer);
                break;
    }
}
```

```
Ejercicios — -bash — 80×9
MacBook-Pro-de-Patricia:Ejercicios patriciasantostabares$ gcc PRO_7.GARCETA14.c

MacBook-Pro-de-Patricia:Ejercicios patriciasantostabares$ ./a.out
El hijo escribe en el pipe...
El padre lee del pie...
El mensaje leido es: Hola papi
MacBook-Pro-de-Patricia:Ejercicios patriciasantostabares$
```

Primero se crea la tubería con pipe(), y a continuación el proceso hijo. Recordemos que cuando se crea un proceso hijo con fork(), recibe una copia de todos los descriptores de ficheros del proceso padre, incluyendo copia de los descriptores de ficheros del pipe (fd[0] y fd[1]). Esto permite que el proceso hijo mande datos al extremo de escritura del pipe fd[1] y el padre los reciba del extremo de lectura fd[0]



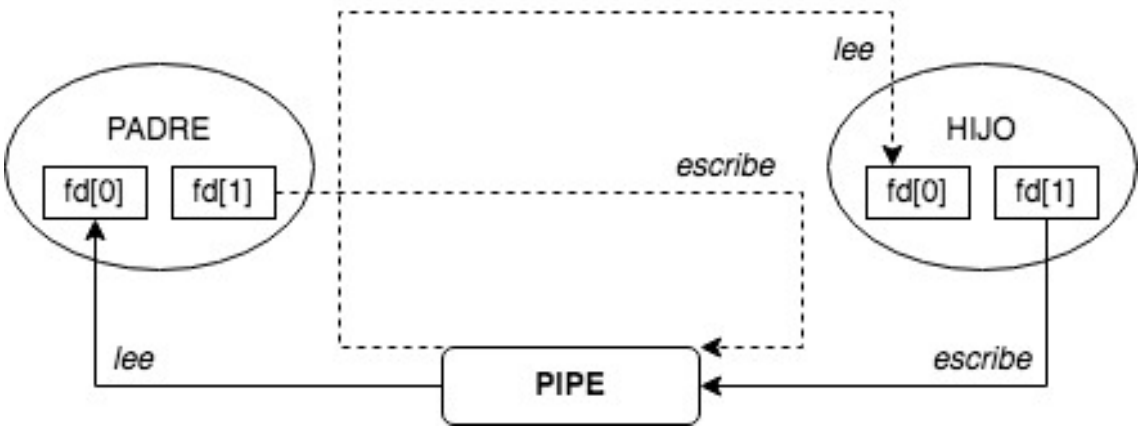

Los procesos padre e hijo están unidos por el pie, pero la comunicación es en una única dirección, por tanto se debe decidir en qué dirección se envía la información, del padre al hijo o del hijo al padre; y dado que los descriptores se comparten siempre, debemos estar seguros de cerrar el extremo que nos interesa.

Cuando el flujo de información va del padre hacia el hijo:

- El padre debe cerrar el descriptor de lectura fd[0]
- El hijo debe cerrar el descriptor de escritura fd[1]

Cuando el flujo de información va del hijo hacia el padre:

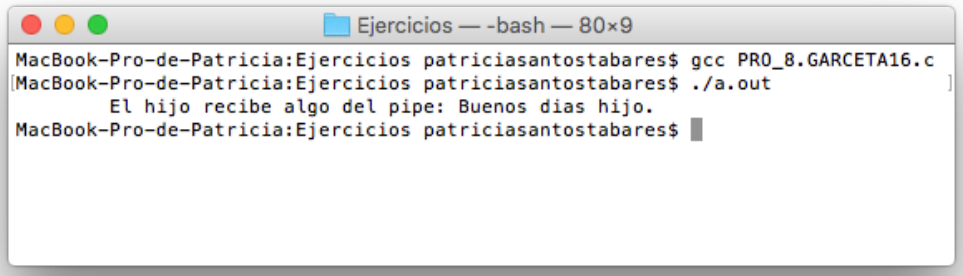
- El padre debe cerrar el descriptor de escritura fd[1]
- El hijo debe cerrar el descriptor de lectura fd[0]



# ACTIVIDAD GUIADA 7.

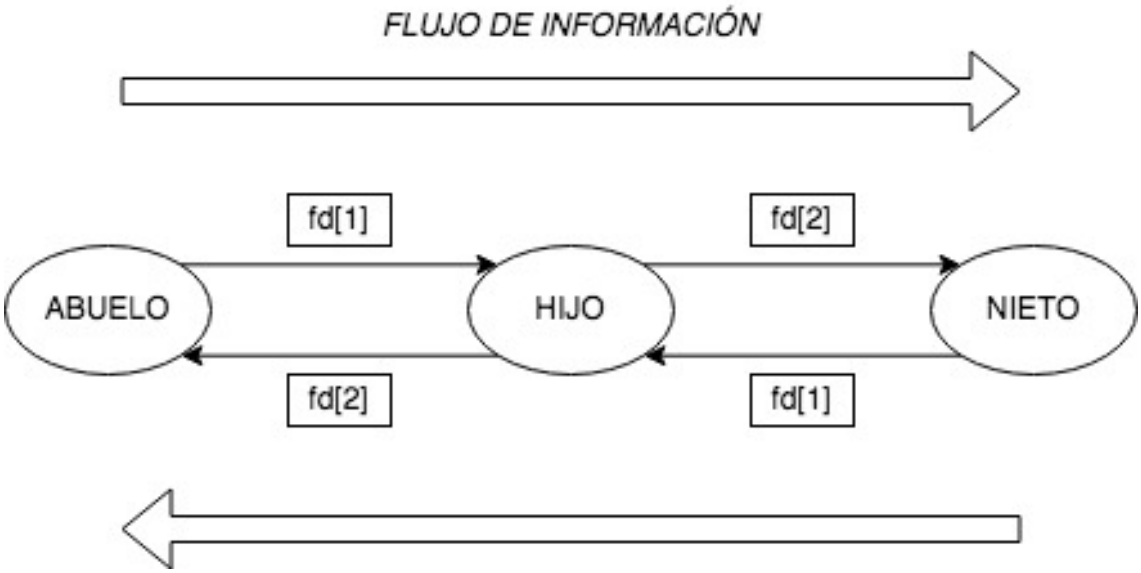
En el siguiente ejemplo crea un pipe en el que el padre envía un mensaje al hijo, el flujo de la información va del padre al hijo, el padre debe cerrar el descriptor fd[0] y el hijo fd[1]; el padre escribe en fd[1] y el hijo lee en fd[0]:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fd[2];
    char buffer[17];
    pid_t pid;
    char saludoPadre[]="Buenos dias hijo.";
    pipe(fd); //Se crea el pipe
    pid=fork(); //Se crea el proceso hijo
    switch(pid)
    {
        case -1: //ERROR
                printf("No se ha podido crear el hijo...");
                exit(-1);
                break;
        case 0: //HIJO RECIBE
                close(fd[1]);
                read(fd[0], buffer, sizeof(buffer)); //Lee del pipe
                printf("\tEl hijo recibe algo del pipe: %s\n", buffer);
                break;
        default: //PADRE ENVIA
                close(fd[0]);
                write(fd[1], saludoPadre, strlen(saludoPadre)); //Escribe en el pipe
                wait(NULL); //Espera al proceso hijo
                break;
    }
    return 0;
}
```

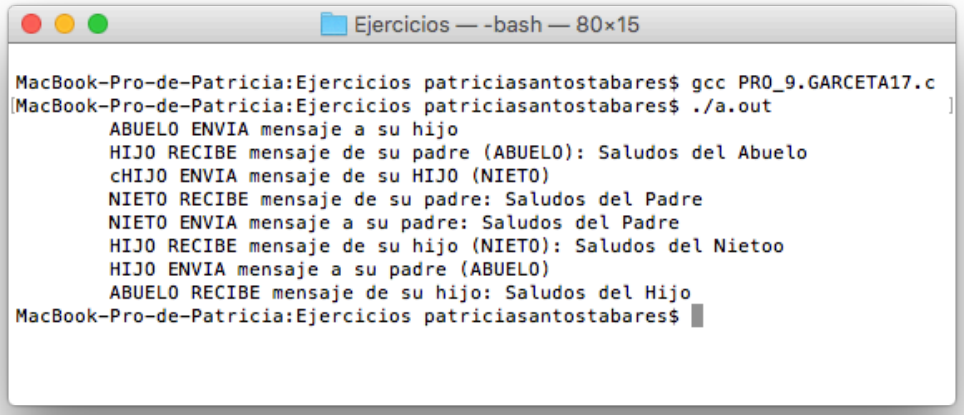


# ACTIVIDAD GUIADA 8.

En el siguiente ejemplo vamos a hacer que padres e hijos puedan enviar y recibir información, como la comunicación es en un único sentido crearemos dos pipes fd1 y fd2. Cada proceso usará un pipe para enviar la información y otro para recibirla. Partimos de los procesos ABUELO, HIJO y NIETO:




- El ABUELO usará fd1 para enviar información al HIJO y recibirá información de éste a través del fd2.
- El HIJO usará el fd2 para enviar información al NIETO y recibirá información de este a través de fd1.
- El NIETO usará fd1 para enviar información al HIJO (su padre) y recibirá información de este a través del fd2.



```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    pid_t pid, hijo_pid, pid2, hijo2_pid;
    int fd1[2];
    int fd2[2];

    char saludoAbuelo[]="Saludos del Abuelo";
    char saludoPadre[]="Saludos del Padre";
    char saludoHijo[]="Saludos del Hijo";
    char saludoNieto[]="Saludos del Nieto";

    char buffer[80]="";
    pipe(fd1);
    pipe(fd2);

    pid=fork(); //Soy el Abuelo, creo a Hijo
    if(pid==-1){
        printf("No se ha podido crear el proceso hijo...");
        exit(-1);
    }
   if(pid==0)
   {
       //Nos encontramos en el proceso hijo
       pid2=fork(); //Soy hijo y creo al Nieto
       switch(pid2)
       {
           case -1:
                       printf("No se ha podido crear el proceso nieto...");
                       exit(-1);
                       break;

           case 0:     //Proceso Hijo del Hijo = NIETO
                       //NIETO RECIBE
                       close(fd2[1]);
                       read(fd2[0], buffer, sizeof(buffer));
                       printf("\tNIETO RECIBE mensaje de su padre: %s\n", buffer);

                       //NIETO ENVIA
                       printf("\tNIETO ENVIA mensaje a su padre: %s\n", buffer);
                       close(fd1[0]);
                       write(fd1[1], saludoNieto, strlen(saludoNieto));
                       break;
```

```
            default:    //Proceso Hijo del abuelo, padre del nieto
                        //HIJO RECIBE
                        close(fd1[1]);
                        read(fd1[0], buffer, sizeof(buffer));
                        printf("\tHIJO RECIBE mensaje de su padre (ABUELO): %s\n", buffer);

                        //HIJO ENVIA A SU HIJO (NIETO)
                        printf("\tcHIJO ENVIA mensaje de su HIJO (NIETO)\n");
                        close(fd2[0]);
                        write(fd2[1], saludoPadre, strlen(saludoPadre));

                        //HIJO RECIBE DE SU HIJO
                        close(fd1[1]);
                        read(fd1[0], buffer, sizeof(buffer));
                        printf("\tHIJO RECIBE mensaje de su hijo (NIETO): %s\n", buffer);

                        //HIJO ENVIA A SU PADRE (ABUELO)
                        printf("\tHIJO ENVIA mensaje a su padre (ABUELO)\n");
                        close(fd2[0]);
                        write(fd2[1], saludoHijo, strlen(saludoHijo));
                        break;
        }
    }
    else
    {
        //Nos encontramos en el proceso padre (ABUELO)
        //ABUELO ENVIA
        printf("\tABUELO ENVIA mensaje a su hijo\n");
        close(fd1[0]);
        write(fd1[1], saludoAbuelo, strlen(saludoAbuelo));
        hijo_pid=wait(NULL);

        //ABUELO RECIBE DE SU HIJO
        close(fd2[1]);
        read(fd2[0], buffer, sizeof(buffer));
        printf("\tABUELO RECIBE mensaje de su hijo: %s\n", buffer);
    }

    return 0;
}
```

## 6.2 Tuberías con nombre o FIFOs

Los pipes vistos anteriormente establecían un canal de comunicación entre procesos emparentados (padre-hijo). Los FIFOS permiten comunicar procesos que no tienen que estar emparentados.

Un FIFO es como un fichero con nombre que existe en el 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. Los FIFOS tienen algunas diferencias con los ficheros:

- Una operación de escritura en un FIFO queda en espera hasta que el proceso pertinente abra el FIFO para iniciar la lectura.
- Solo se permite la escritura de información cuando un proceso vaya a recoger dicha información.

Hay varias formas de crear un FIFO: ejecutando el comando mknod desde la línea de comandos de Linux o desde un programa C usando la función mknod(). Su formato es:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t modo, dev_t dev);
```

Donde

**pathname** es el nombre del dispositivo creado.
**modo** especifica tanto los permisos de uso y el tipo de nodo que se creará. Debe ser una combinación de uno de los tipos de fichero que se enumeran a continuación:

- S_IFREG o 0: para especificar un fichero normal (que será creado vacío).
- S_IFCHR: para especificar un fichero especial de caracteres.
- S_IFBLK: un fichero especial de bloques.
- S_IFIFO: para crear un FIFO.

# ACTIVIDAD GUIADA 9.

A continuación se muestra un ejemplo de uso de FIFOS. El programa fifocrea.c crea un FIFO de nombre FIFO2 y lee la información del FIFO; mientras no haya información quedará en espera. El programa fifoescribe.c escribe información en el FIFO.

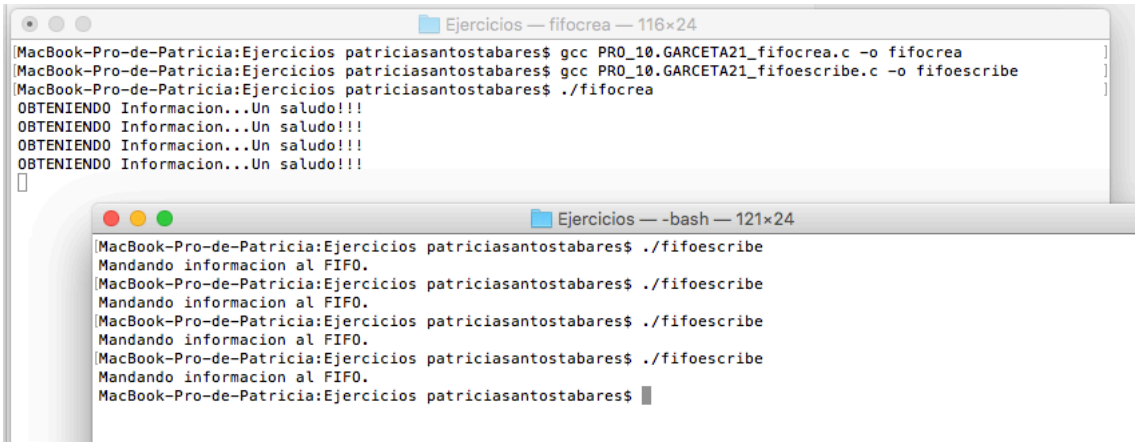
En el terminal se ejecutará primero fifocrea y después varias veces fifoescribe desde otro terminal.

**FIFOCREA.C**

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fp;
    int p, bytesleidos;
    char saludo[]="Un saludo!!!\n";
    char buffer[10];

    p=mknod("FIFO2", S_IFIFO|0666, 0); //PERMISO DE LECTURA Y ESCRITURA

    if(p==-1){
        printf("Ha ocurrido un error...\n");
        exit(0);
    }
    while(1)
    {
        fp=open("FIFO2", 0);
        bytesleidos=read(fp, buffer, 1);
        printf("OBTENIENDO Informacion...");
        while(bytesleidos!=0){
            printf("%s", buffer);
            bytesleidos=read(fp, buffer, 1);
        }
        close(fp);
    }
    return 0;
}
```

## FIFOESCRIBE.C

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>

int main()
{
    int fp;
    char saludo[]="Un saludo!!!\n";
    fp=open("FIFO2", 1);

    if(fp==-1){
        printf("Error al abrir el fichero...");
        exit(1);
    }

    printf("Mandando informacion al FIFO.\n");
    write(fp, saludo, strlen(saludo));
    close(fp);

    return 0;
}
```



## 6.3 Sincronización entre procesos (SEÑALES)

Para que los procesos interactúen unos con otros necesitan cierto nivel de sincronización, es decir, es necesario que haya un funcionamiento coordinado entre los procesos a la hora de ejecutar alguna tarea. Podemos utilizar señales para llevar a cabo la sincronización entre dos procesos.

A continuación se muestran una serie de funciones útiles que utilizaremos para que un proceso padre y otro hijo se comuniquen de forma síncrona utilizando señales.

Una señal es como un aviso que un proceso manda a otro proceso. **La función signal() es el gestor de señales por excelencia que especifica la acción que debe realizarse cuando un proceso recibe una señal.** Su formato es el siguiente:

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

Recibe dos parámetros:

- **Señal**: contiene el número de señal que queremos capturar. En nuestro ejemplo pondremos SIGUSR1 que es una señal definida por el usuario para ser usada en programas de aplicación. Otra señal interesante es SIGKILL que se usa para terminar con un proceso.
- **Func**: contiene la función a la que queremos que se llame. Esta función es conocida como el manejador de la señal (**signal handler**). En el ejemplo que se verá a continuación se definen dos manejadores de señal, uno para el proceso padre (void gestion_padre (int segnal)) y otro para el hijo (void gestion_hijo(int segnal)).

La función devuelve un puntero al manejador previamente instalado para esa señal. Un ejemplo de uso de la función: signal (SIGUSR1, gestion_padre): significa que cuando el proceso (en este caso el proceso padre) recibe una señal SIGUSR1 se realizará una llamada a la función gestion_padre().

**Para enviar una señal usaremos la función kill()**

```
#include <signal.h>
int kill(int Pid, int señal));
```

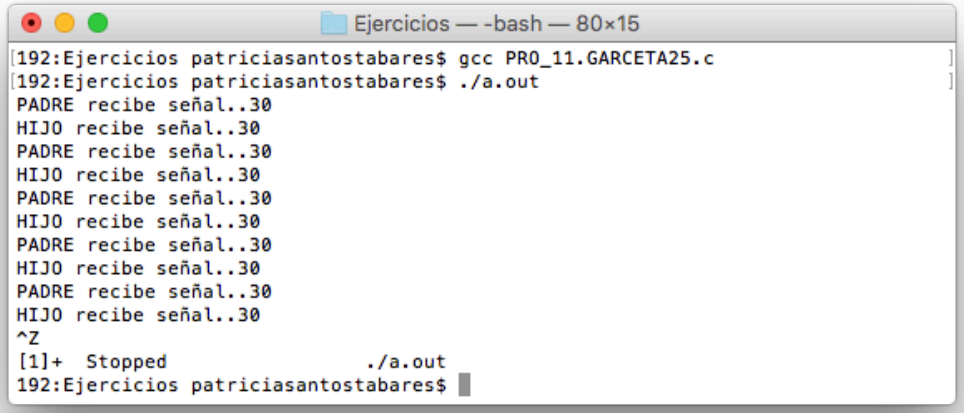
Recibe dos parámetros: el PID del proceso que recibirá la señal y la señal. Por ejemplo y suponiendo que pid_padre es el PID de un proceso padre: kill(pid_padre, SIGUSR1); envía una señal SIGUSR1 al proceso padre.

**Cuando queremos que un proceso espere a que le llegue una señal, usamos la función pause().** Para capturar esa señal, el proceso debe haber establecido un tratamiento de la misma con la función signal(). Este es su formato: int pause(void).

**Por último, la función sleep() suspende al proceso que realiza la llamada la cantidad de segundos indicada o hasta que se reciba una señal.**

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

```c
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <signal.h>

void gestion_padre(int segnal){
    printf("PADRE recibe señal..%d\n", segnal);
}
void gestion_hijo(int segnal){
    printf("HIJO recibe señal..%d\n", segnal);
}

int main()
{
    int pid_padre, pid_hijo;
    pid_padre = getpid();
    pid_hijo = fork();

    switch(pid_hijo)
    {
        case -1:printf("Error al crear el proceso hijo...\n");
                exit(-1);

        case 0: signal(SIGUSR1, gestion_hijo);
                while(1){
                    sleep(1);
                    kill(pid_padre, SIGUSR1);
                    pause();
                }
                break;

        default:signal(SIGUSR1, gestion_padre);
                while(1){
                    pause();
                    sleep(1);
                    kill(pid_hijo, SIGUSR1);
                }
                break;
    }
    return 0;
}
```




# 7. PROGRAMACIÓN DE PROCESOS EN JAVA

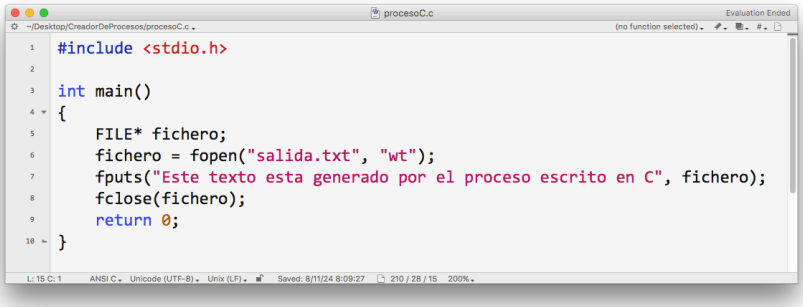
## ACTIVIDAD GUIADA 1.

Desde un programa escrito en Java, ejecutar un programa escrito y compilado en C indicando el directorio de trabajo. El programa en C debe escribir un texto en un fichero. Dicho fichero debe guardarse en el directorio de trabajo indicado.

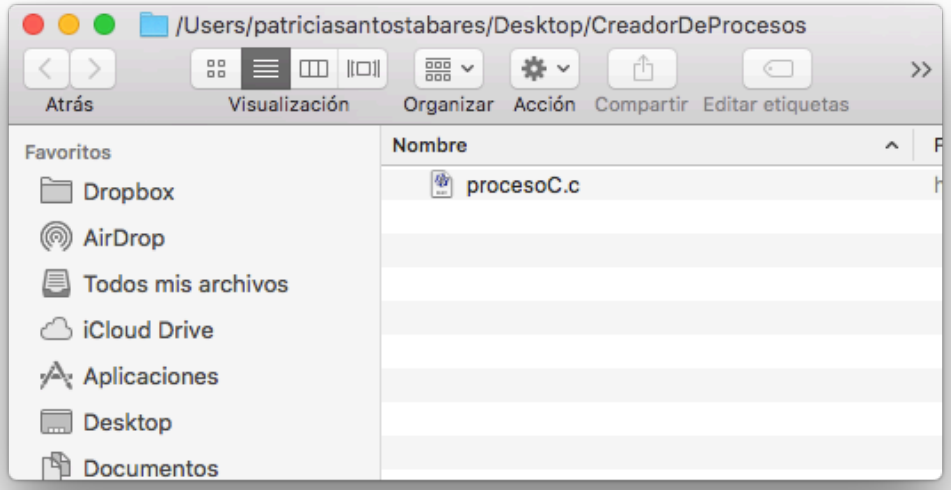
*Proceso iniciador (Java) Fichero Creador de Procesos con ProcessBuilder*

```
package creadorprocesosprocessbuilder;
import java.io.File;
import java.io.IOException;

public class CreadorProcesosProcessBuilder {

    public static void main(String[] args) {
        try {
            ProcessBuilder pBuilder = new ProcessBuilder("/Users/patriciasantostabares/Desktop/CreadorDeProcesos/procesoC.out");
            pBuilder.directory(new File("/Users/patriciasantostabares/Desktop/CreadorDeProcesos"));
            Process proceso = pBuilder.start();
            int valorRetorno = proceso.waitFor();
            if(valorRetorno == 0)
                System.out.println("El proceso se ha completado satisfactoriamente");
            else
                System.out.println("El proceso ha fallado. Código de error = "+valorRetorno);
        } catch(IOException | InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

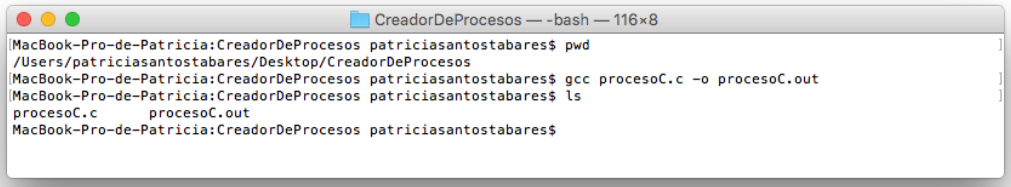
*Proceso iniciado (C)*



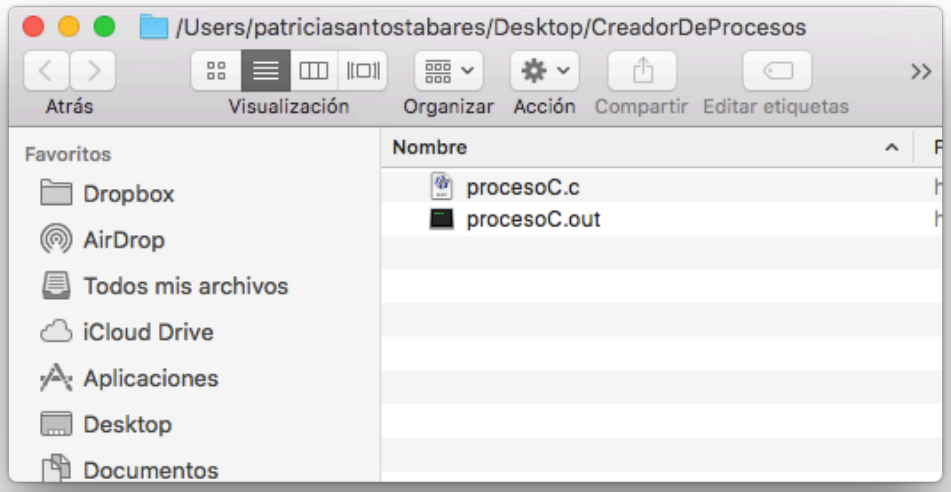
*Carpeta "CreadorDeProcesos" en la que se encuentra mi programa en lenguaje C*



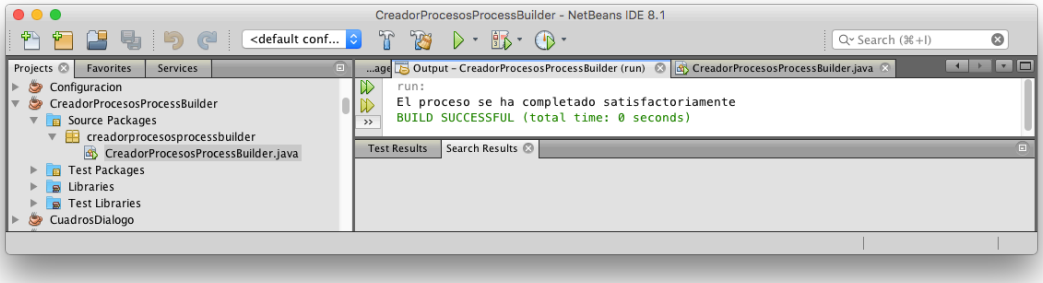

*Compilo procesoC.c*



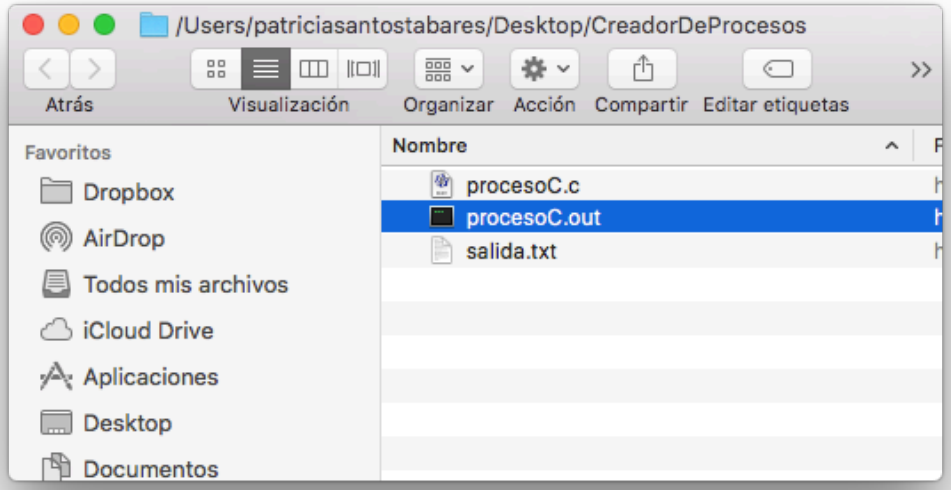

*Compruebo que he obtenido el ejecutable "procesoC.out"*



*Compiloy ejecuto  CreadorProcesoProcessBuilder.java*




*Compruebo que se ha ejecutado procesoC y ha creado el fichero salida.txt*




*Compruebo el contenido del fichero salida.txt*