

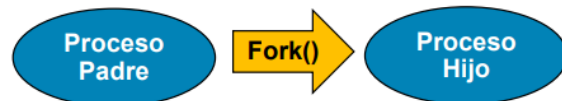
UT1 – Programación Multiproceso

PID → id del proceso suele ser hijo

PPID → id del proceso padre

UNIX/LINUX

fork: crea un nuevo proceso



Al crear un proceso hijo se crea un bloque de control de proceso (BCP) idéntico al padre solo cambiar el PID

para diferenciar un programa hijo de uno padre, el valor del PID que devolverá fork será 0 para el hijo y cualquier número diferente a 0 sería el padre

Programas para instalar en linux

`sudo apt-get update`

`sudo apt install gcc` – compilador de gcc

para ver `gcc --version`

para compilar `pcc fuente.c -o ejecutable`

```
alumno@alumnov:~/ejercicios$ gcc ps1.c -o ps1
alumno@alumnov:~/ejercicios$
```

para ejecutar `./fuente`

```
alumno@alumnov:~/ejercicios$ ./ps1
PID del proceso actual es: 9118
PID del proceso padre es: 3312
```

El programa al ejecutarse muchas veces, cambia el PID del proceso hijo pero del proceso padre siempre es el mismo

```

alumno@alumnomv:~/ejercicios$ ./ps1
PID del proceso actual es: 9118
PID del proceso padre es: 3312
alumno@alumnomv:~/ejercicios$ ./ps1
PID del proceso actual es: 9121
PID del proceso padre es: 3312
alumno@alumnomv:~/ejercicios$ ./ps1
PID del proceso actual es: 9122
PID del proceso padre es: 3312
alumno@alumnomv:~/ejercicios$ ./ps1
PID del proceso actual es: 9123
PID del proceso padre es: 3312
alumno@alumnomv:~/ejercicios$ ./ps1
PID del proceso actual es: 9124
PID del proceso padre es: 3312
alumno@alumnomv:~/ejercicios$

```

2.4. Comunicación entre procesos

Mecanismos que permiten a los procesos comunicarse y sincronizarse entre ellos.

Diferentes mecanismos:

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

pipes:

pipe comunica 2 procesos mediante un mecanismo half-duplex

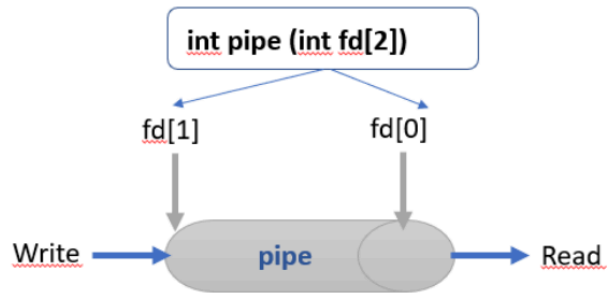


Cuando proceso quiere leer del **pipe** y está **vacío** -> **se bloquea a la espera datos**

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

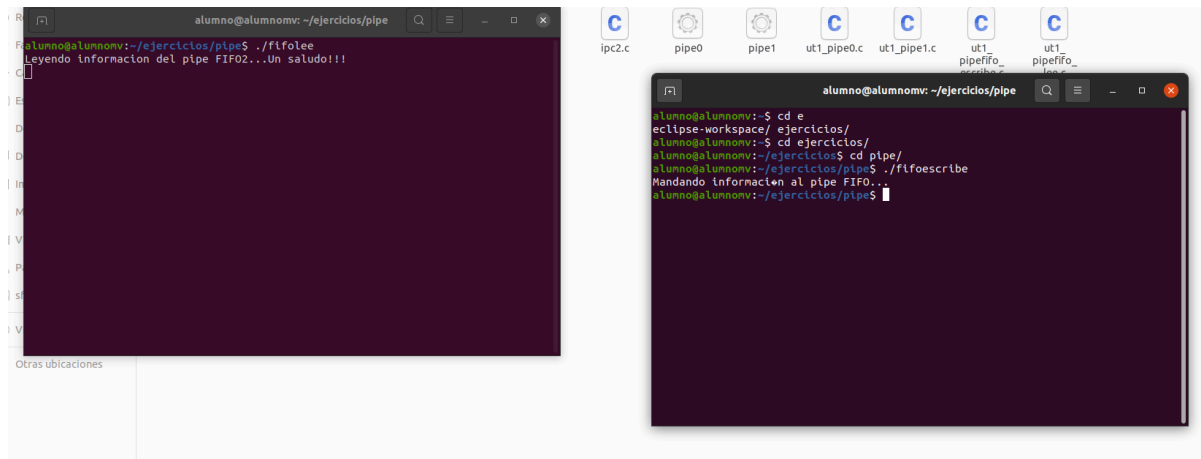
Función **pipe** permite crear el pipe:

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



PIPO lee- PIPO escribe

- se necesitan abrir 2 terminales para abrir este fichero.
- para este tipo de ejercicio no se necesita el **fork**



2.4.3 Señales

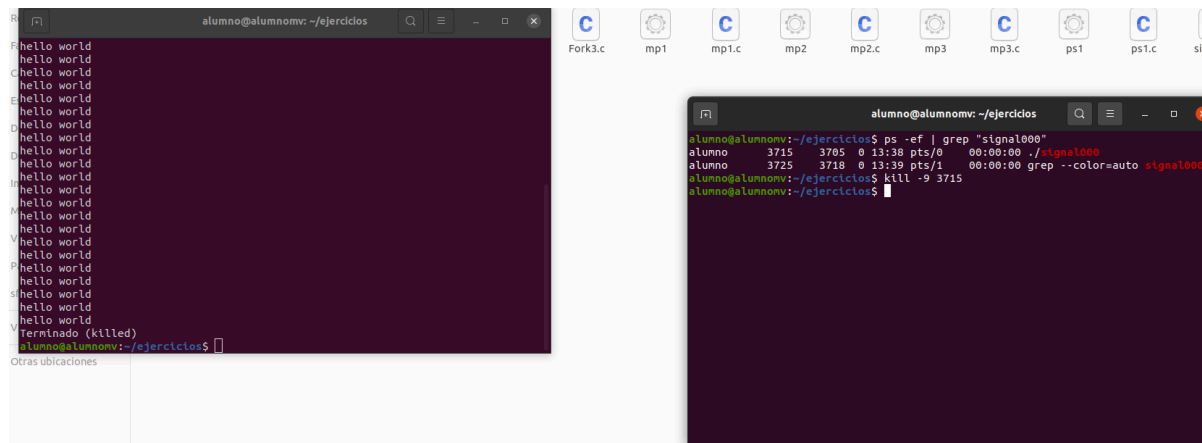
para terminar un proceso:

ps es para saber los datos de la señal y **grep** es para filtrar.

Kill -9 es para dar la señal de matar el proceso, y 3715 es el numero del proceso

```
alumno@alumnov:~/ejercicios$ ps -ef | grep "signal000"
alumno    3715    3705  0 13:38 pts/0    00:00:00 ./signal000
alumno    3725    3718  0 13:39 pts/1    00:00:00 grep --color=auto signal000
alumno@alumnov:~/ejercicios$ kill -9 3715
alumno@alumnov:~/ejercicios$
```

Proceso terminado



Nota: esto es un programa que sirve para poner “hello world” de manera infinita, lo normal seria que con ctrl+c terminaria el proceso, pero en el codigo esta puesto que no sea capaz de terminar, en esto caso se pondría “-9” que aunque en el codigo se diga que ignore “-9” se termine ya que no se puede ignorar

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

```
alumno@alumnov:~/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
```

1. Programar signal1.c que muestre por pantalla la fecha y la hora a la que se inicia el proceso junto con su pid. Al recibir la señal SIGINT deberá mostrar la fecha y hora de finalización del programa. ./signal1
 Inicio del proceso 71887: 30/11/2023 19:19:23 ^CFin del proceso 71887: 30/11/2023 19:19:30

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

```

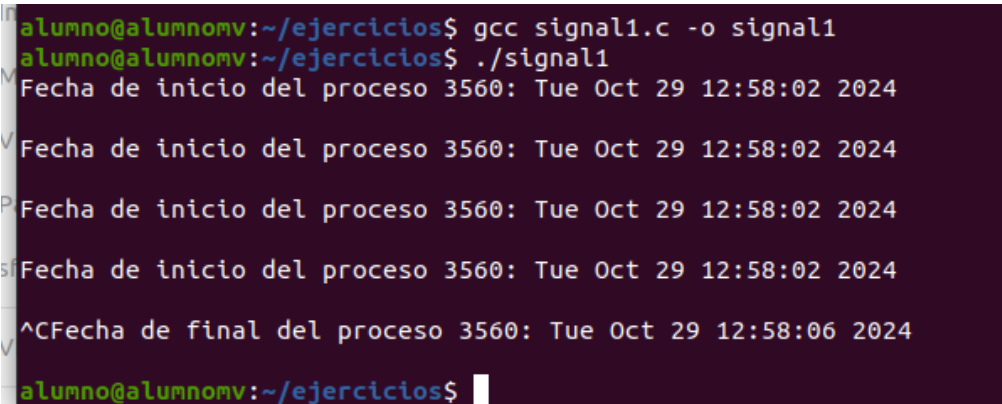
//variables para almacenar la fecha y hora
time_t i;
time_t f;

void sig_handler(int signo)
{
    f=time(NULL);
    if (signo == SIGINT){
        printf("Fecha de final del proceso %d: %s\n",
getpid(),ctime(&f));
        exit(0);
    }
}

int main(void)
{
    i=time(NULL);
    signal(SIGINT, sig_handler);

    //Bucle infinito proceso queda a la espera de la señal
    while(1){
        printf("Fecha de inicio del proceso %d: %s\n",getpid(),
ctime(&i));
        sleep(1);
    }
    return 0;
}

```



```

alumno@alumnov:~/ejercicios$ gcc signal1.c -o signal1
alumno@alumnov:~/ejercicios$ ./signal1
Fecha de inicio del proceso 3560: Tue Oct 29 12:58:02 2024
Fecha de inicio del proceso 3560: Tue Oct 29 12:58:02 2024
Fecha de inicio del proceso 3560: Tue Oct 29 12:58:02 2024
Fecha de inicio del proceso 3560: Tue Oct 29 12:58:02 2024
^CFecha de final del proceso 3560: Tue Oct 29 12:58:06 2024
alumno@alumnov:~/ejercicios$

```

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

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

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

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

```
int main() {
```

```
    int pipefd[2];
```

```
    pid_t pid;
```

```
    int num, result;
```

```
    // Crear el pipe
```

```
    if (pipe(pipefd) == -1) {
```

```
        perror("Pipe failed");
```

```
        return 1;
```

```
    }
```

```
    pid = fork();
```

```
    if (pid == -1) {
```

```
        perror("Fork failed");
```

```
        return 1;
```

```
}
```

```
if (pid == 0) {
```

```
    // Proceso hijo
```

```
    close(pipefd[1]); // Cerrar el descriptor de escritura
```

```
    read(pipefd[0], &num, sizeof(num)); // Leer el número del pipe
```

```
    result = num + 1; // Incrementar el número
```

```
        write(pipefd[0], &result, sizeof(result)); // Enviar el resultado al  
padre
```

```
    close(pipefd[0]);
```

```
    exit(0);
```

```
} else {
```

```
    // Proceso padre
```

```
    close(pipefd[0]); // Cerrar el descriptor de lectura
```

```
    printf("Introduce un numero: ");
```

```
    scanf("%d", &num);
```

```
    write(pipefd[1], &num, sizeof(num)); // Escribir el número en el pipe
```

```
    wait(NULL); // Esperar a que termine el hijo
```

```
    read(pipefd[0], &result, sizeof(result)); // Leer el resultado del hijo
```

```
    printf("Número incrementado: %d\n", result);
```

```
    close(pipefd[1]);
```

```
}
```

```
return 0;
```

```
}
```

Programación Concurrente:

UT2 – Programación Multihilo

Thread (Hilos)

Unidad mínima de ejecución que se permite en un procesador, un hilo es un mismo proceso pero que se va ejecutando en tiempos diferentes

- Los hilos no pueden ejecutarse ellos solos, necesitan la supervisión de un proceso padre para ejecutarse. Por ejemplo, en java, al lanzar un proceso se crea un Thread que ejecuta el main

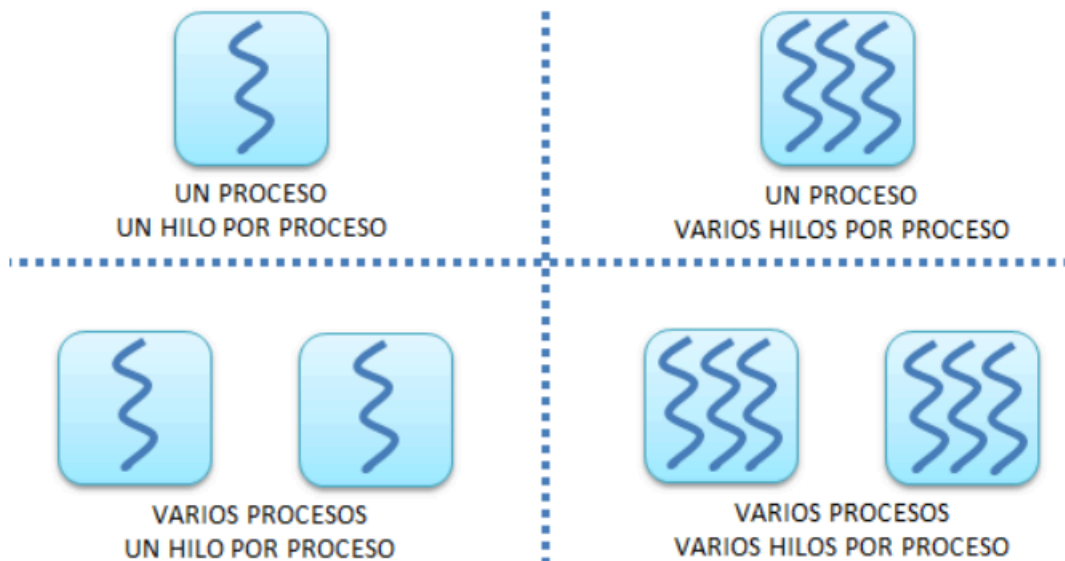


UN PROCESO
UN HILO POR PROCESO



UN PROCESO
VARIOS HILOS POR PROCESO

- ❖ Los hilos se ejecutan dentro del contexto de un programa.
- ❖ Los procesos tienen espacios de memoria independientes.
- ❖ Los hilos que dependen de un mismo proceso, comparten la memoria del proceso entre ellos.
- ❖ Un programa en ejecución (proceso) puede realizar distintas tareas (threads) al mismo tiempo.



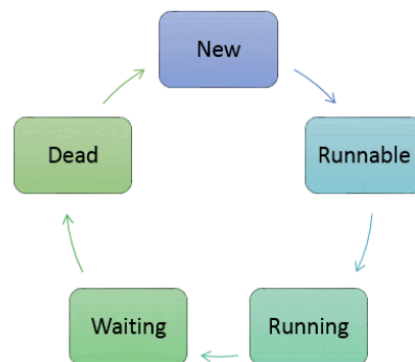
1.2 Estados de hilos

❖ **Nuevo (New):** el hilo está preparado, pero todavía no se ha hecho la llamada de ejecución. Los hilos se inicializan en la creación del proceso

❖ **Listo (Runnable):** el proceso está listo para entrar en ejecución en cuanto el SO lo decida.

❖ **Ejecutando (Running):** el hilo está en ejecución.

❖ **Bloqueado (Waiting/Timed Waiting):** hilo bloqueado indefinidamente a la espera suceda algún evento o bloqueado por un tiempo determinado



❖ **Terminado (Dead/Terminated):** ha finalizado su ejecución. El hilo no libera recursos ya que no les pertenecen (son del proceso)

Los hilos se crean a nivel usuario(programados), el sistema operativo solo ve procesos.

Ejemplos de clase

```

alumno@alumnomv:~/ejercicios/Hilos$ javac Hilo1.java
alumno@alumnomv:~/ejercicios/Hilos$ java Hilo1
Ejecutando dentro del Hilo... 0
Ejecutando dentro del Hilo... 1
Ejecutando dentro del Hilo... 2
Ejecutando dentro del Hilo... 3
Ejecutando dentro del Hilo... 4
Ejecutando dentro del Hilo... 5
Ejecutando dentro del Hilo... 6
Ejecutando dentro del Hilo... 7
Ejecutando dentro del Hilo... 8

```

En los hilos para que sigan un orden al terminar se usa `join()` para asegurar que un proceso termine

```

alumno@alumnomv:~/ejercicios/Hilos$ javac HiloContador.java
alumno@alumnomv:~/ejercicios/Hilos$ java HiloContador
Hilo Contador 1: 0
Hilo Contador 1: 37
Hilo Contador 1: 38
Hilo Contador 1: 39
Hilo Contador 1: 40
Hilo Contador 1 finalizado
Fin del programa
alumno@alumnomv:~/ejercicios/Hilos$

```

en este caso “fin del programa” podría aparecer al principio o en medio si no usamos `join`

```

public static void main(String[] args) {

    //Creamos los objetos
    HiloContador c1 = new HiloContador("Contador 1", 40);
    HiloContador c2 = new HiloContador("Contador 2", 50);
    HiloContador c3 = new HiloContador("Contador 3", 20);
    HiloContador c4 = new HiloContador("Contador 4", 70);

    //Creamos los hilos
    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);
    Thread t3 = new Thread(c3);
    Thread t4 = new Thread(c4);

    //Iniciamos los hilos
    t1.start();
    t2.start();
    t3.start();
    t4.start();

    try{
        t1.join();
        t2.join();
        t3.join();
        t4.join();
    }
    catch(Exception ex){System.out.println(ex);}

    //Fin
    System.out.println("Fin del programa");

}

```

Resumen: se usa join para mantener un orden ya que incluso el programa principal seria un hilo, en este ejercicio creamos 4 hilos pero el mismo programa en si seria un 5 hilo, y "Fin del programa" que seria lo último en ejecutarse se ejecutaría en cualquier parte del codigo.