# Curso de programación en C moderno

(II Edición)

Neira Ayuso, Pablo  Falgueras García, Carlos

*Tema 16*

# Fork

# Índice

# Paralelismo



- A veces es *necesario* que nuestro programa realice varias tareas **a la vez**
- Otras veces es *interesante* dividir el trabajo en partes independientes que pueden realizarse simultáneamente (**en paralelo**). Esto puede mejorar enormemente la eficiencia de nuestro programa, incluso si nuestro hardware no tiene la capacidad de ejecutar varios procesos a la vez
- **No siempre es lo mejor**. Paralelizar un programa tiene un coste de rendimiento que, a veces, puede superar a la mejora obtenida.

# Procesos



- En Unix, hilos y procesos son esencialmente lo mismo
- Todo proceso tiene asociado un número que lo identifica: **PID**(*Process IDentification*)
- En Unix todos procesos se crean mediante la llamada al sistema `pid_t fork(void);`

# fork()



- En el momento en que un programa llama a `fork()`, se duplica.
- A partir de entonces existirán dos procesos (padre e hijo) con el mismo código, las mismas variables, etc.
- El proceso hijo tendrá una copia de la memoria del padre en el instante en el que se llamó a `fork()`.
- La memoria es independiente. Si uno de los procesos modifica una variable, no afecta a la variable del otro proceso. (Existe la opción de que ambos procesos compartan la memoria, pero se excede del propósito de este curso)

# Ejemplo



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

int main()
{
   int pid;
   int var = 0;

   pid = fork();
   if (pid == -1) {
      perror("fork");
      exit(EXIT_FAILURE);
   }
   if (pid == 0) {
      printf("Soy el hijo\n\tPID = %d\n",
             getpid());

      var = 10;
      printf("var del hijo = %d\n", var);
   } else {
      printf("Soy el padre\n\tPID = %d\n\tPID hijo = %d\n", getpid(),
           pid);

      wait(NULL); // Esperamos a que termine el hijo
      printf("var del padre = %d\n", var);
   }

   return EXIT_SUCCESS;
}
```

# Ejemplo



```
Soy el hijo
  PID = 7590
Soy el padre
  PID = 7589
  PID hijo = 7590
var del padre = 0
var del hijo = 10
```

# Zombies



- Cuando un proceso hijo muere, no muere del todo. Permanece en un estado llamado **zombie** que permite al padre recuperar información de él aun cuando ha terminado (el código de error por ejemplo)
- El sistema puede llegar a colapsarse si se crean mucho procesos zombies. Conclusión: Siempre llamar a `wait()`)
- Si un proceso padre muere antes que su hijo (no ha llamado a (`wait()`), el comportamiento depende del SO:
  - El padre del proceso hijo pasa a ser el proceso **init** (PID = 1)
  - El hijo es matado también inmediatamente

# Señales



- Son un medio para que los procesos puedan interaccionar entre sí. Un proceso envía una señal y el otro la recibe.
- Existen muchos tipos de señales distintas (`man 7 signal`)
- Para reaccionar ante la recepción de una señal debemos instalar un **callback**

# Ejemplo



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

void sigint_hdl(int sig)
{
  static int cnt = 0;
  if (cnt < 2) {
    printf("No quiero morir!!\n");
    cnt++;
  } else{
    printf("Vaaaleee, me muero\n");
    exit(EXIT_SUCCESS);
  }
}

int main()
{
  struct sigaction sa;
  sa.sa_handler = sigint_hdl;
  sigemptyset(&sa.sa_mask);

  if (sigaction(SIGINT, &sa, NULL) == -1) {
    perror("sigaction");
    return EXIT_FAILURE;
  }

  while(1); // No quiero morir
  return EXIT_SUCCESS;
}
```