Fundamentos de Software de Comunicaciones

Tema 2

Programación del Sistema Operativo

(2ª parte)

Contenidos

- Concurrencia
- Procesos
  - Clonación de procesos: la llamada al sistema fork()
  - Mutación de procesos: la familia de funciones exec()
  - Zombies y huérfanos
- Comunicación entre procesos
  - Tuberías sin nombre (pipes)
  - Tuberías con nombre (FIFOs)

Programas, procesos y hebras

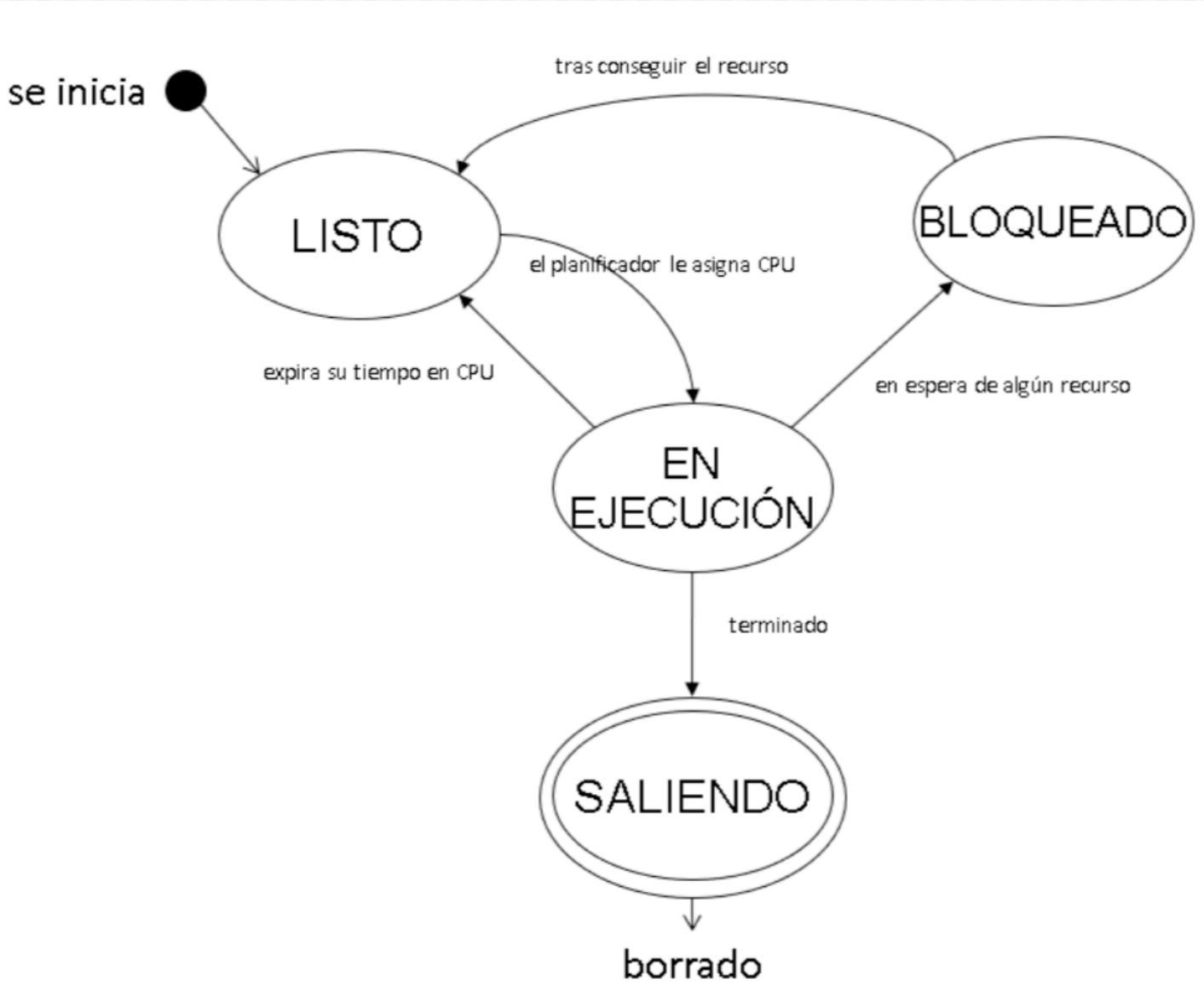
- Cuando compilamos un programa, se genera un ejecutable → está almacenado en disco

- Un proceso es un programa en ejecución, que incluye:
  - El ejecutable cargado en memoria
  - Datos
  - Recursos del sistema, como ficheros
  - Un contexto de seguridad (usuario asociado, permisos)
  - Una o más hebras de ejecución
  - Un ordenador virtual
    - Abstracción fundamental en Linux
    - Un proceso ve al sistema (CPU + memoria) como si fuese exclusivo para él
    - El núcleo se encarga de forma transparente de compartir los recursos entre todos los procesos existentes → concurrencia

Concurrencia

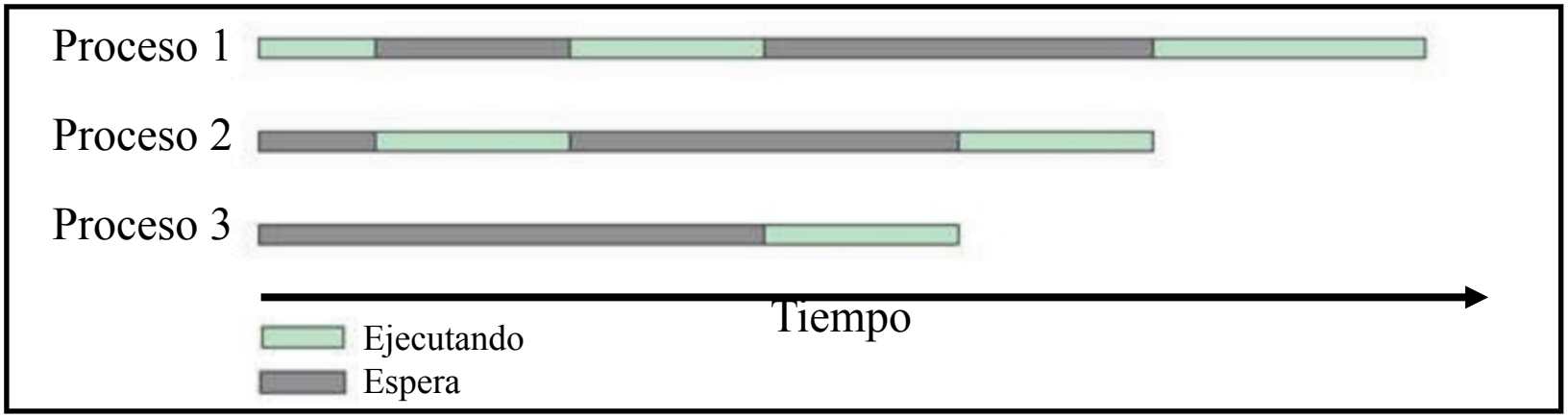
- En un entorno multitarea como UNIX los procesos se ejecutan concurrentemente
  - Habiendo más procesos que CPUs
- El S.O. tiene un planificador que selecciona qué proceso ocupa la CPU en cada momento y cuando debe salir para dejar paso a otro
  - Un proceso no tiene por qué ocupar completamente la CPU hasta que termine
  - Se asignan ranuras de tiempo (time slices) a cada proceso
- Aspecto crítico en el software de comunicaciones
  - Implementación de servicios que atienen a múltiples conexiones

El ciclo de vida de un proceso



Programación concurrente

- Varios procesos pueden realizar trabajos cooperativos, alternándose en la/s CPU/s.
  - Un proceso se dedica a gestionar las solicitudes de nuevos clientes (es su única tarea)
  - El servicio concreto a un cliente lo realiza otro proceso especializado
  - Muchos clientes -> muchos procesos de servicio



- Existen dos métodos para conseguir la concurrencia en nuestros programas:
  - Multitarea pesada: generación de procesos *hijos* a través de fork()
  - Multitarea ligera: generación de hilos de ejecución (*hebras*) concurrentes al principal

Procesos

- Tienen un identificador único que no cambia mientras se ejecuta, el *ProcessID* o *pid*
- El primer proceso que se ejecuta cuando se inicia el sistema se llama *init* y tiene pid 1
- Todos los procesos, excepto el *init*, se generan a partir de otro proceso, por tanto tienen un padre
- Esto define una jerarquía
  - Ver la herramienta **pstree**
- Operaciones con procesos en Unix/Linux
  - Crear un nuevo proceso: **clonar**
  - Cargar un programa existente y ejecutarlo: **mutar**

Procesos

- El identificador de un proceso en el S.O. se obtiene con la llamada a getpid()
- El identificador del proceso padre en el S.O. se obtiene con la llamada a getppid()

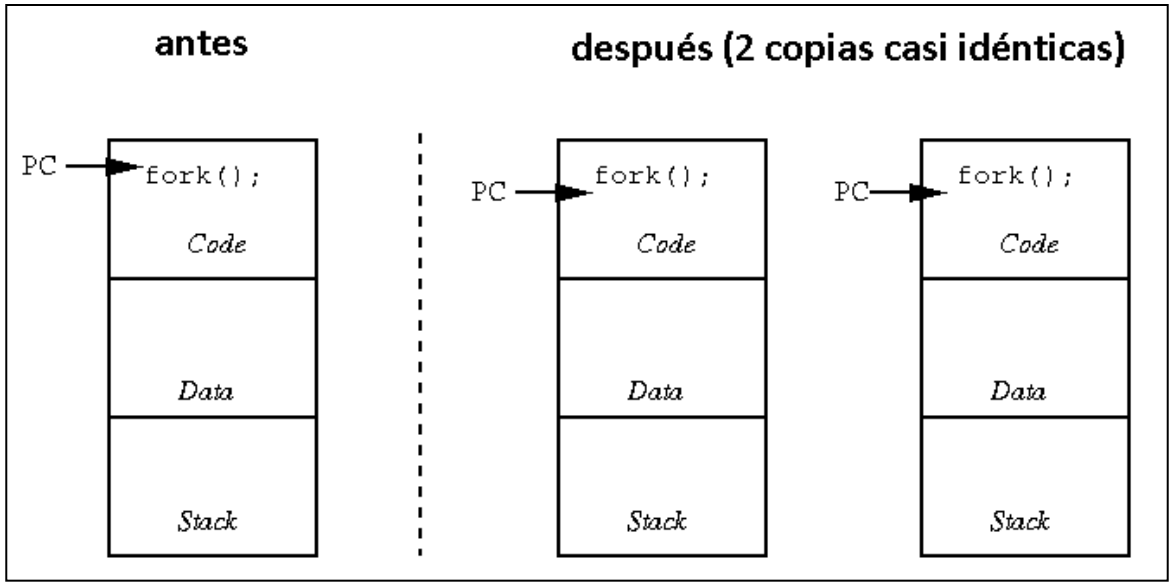
```
#include <sys/types.h> /*para pid_t*/
#include <unistd.h>    /*para getpid y getppid*/

pid_t mi_pid = getpid ();
pid_t padre_pid = getppid();
```

Multiproceso con fork()

- Esta llamada al sistema crea un nuevo proceso hijo, que es una copia idéntica del padre que se está ejecutando (clonar):

```
pid_t  valor = fork();
```

- El hijo recibe una copia del segmento de datos del padre, obteniendo copias de variables y ficheros abiertos previos al fork. ¡Después de la llamada, cada cual mantiene privados sus datos!



Multiproceso con fork()

- Clonar duplica el proceso original. Se copia
  - El contenido de la pila y el montículo
  - El segmento de código y las variables globales (segmento bss)
  - La tabla de descriptores de ficheros completa, por lo que el clon tiene acceso a los ficheros a través de los mismos descriptores abiertos en el original
  - Contador de programa (la sentencia por la que va la ejecución)

- Implicaciones adicionales
  - Ambos procesos tienen el mismo código, por lo que ejecutarán las mismas instrucciones
  - Las variables se copian, pero una vez se clona, las modificaciones son locales a cada proceso
  - Al clonar, el clon pasa a la lista de procesos LISTOS, pero también puede pasar ahí el proceso original, por lo que **nunca se puede asumir el orden en el que se van a ejecutar**

Multiproceso con fork()

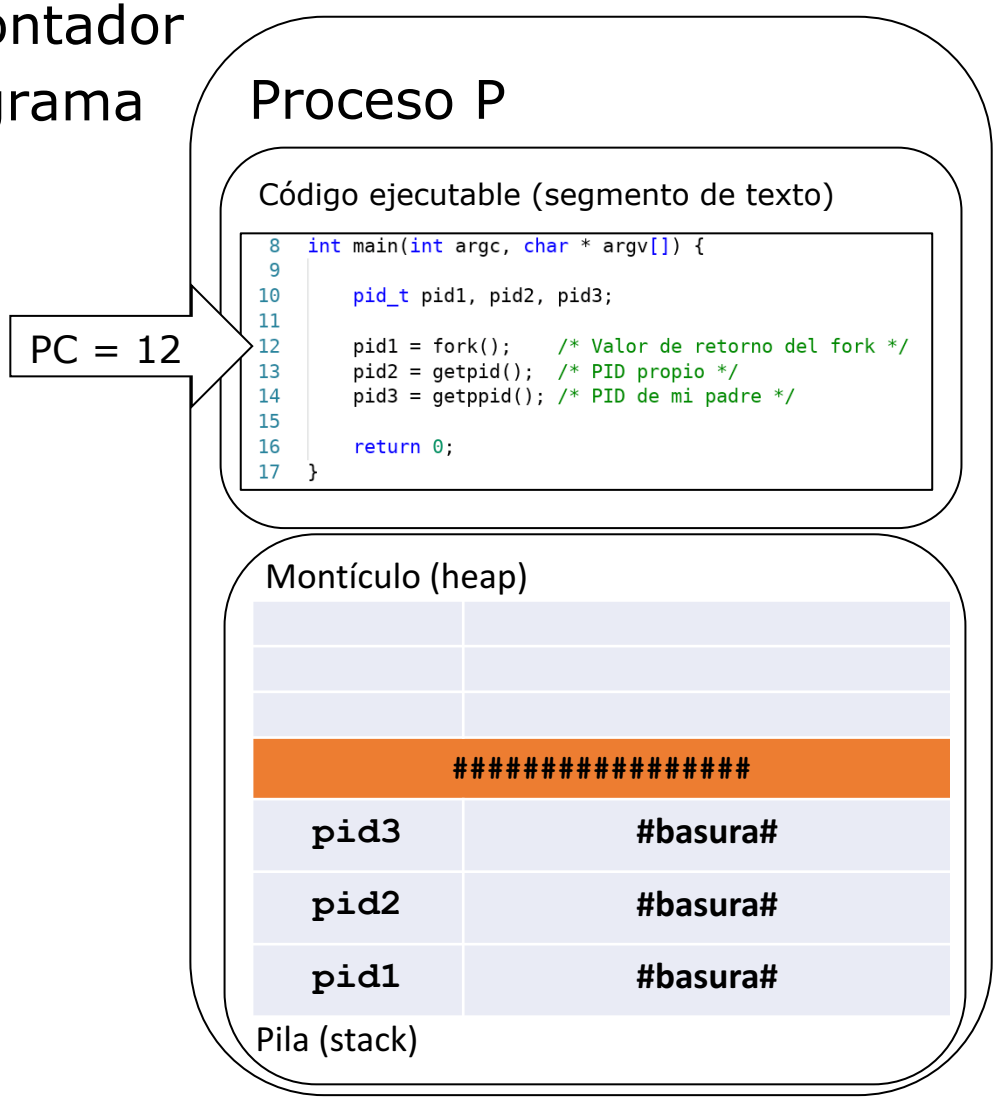
- Valores devueltos:
  - Si hay algún error, devuelve **-1**
  - En caso contrario, **es la única llamada al sistema que devuelve dos valores**:
    - Si estamos en el hijo, fork() devuelve **cero**
    - Si estamos en el padre, fork() devuelve un **valor mayor que cero** (el pid del hijo)

- Diferencias entre padre e hijo
  - Los pids, cada uno tiene uno asignado por el sistema: getpid() devolverá valores diferentes
  - El pid del padre en el hijo (**getppid()**) que se fija al del padre
  - Las señales pendientes, se eliminan
  - Otros (estadísticas, locks, etc.).

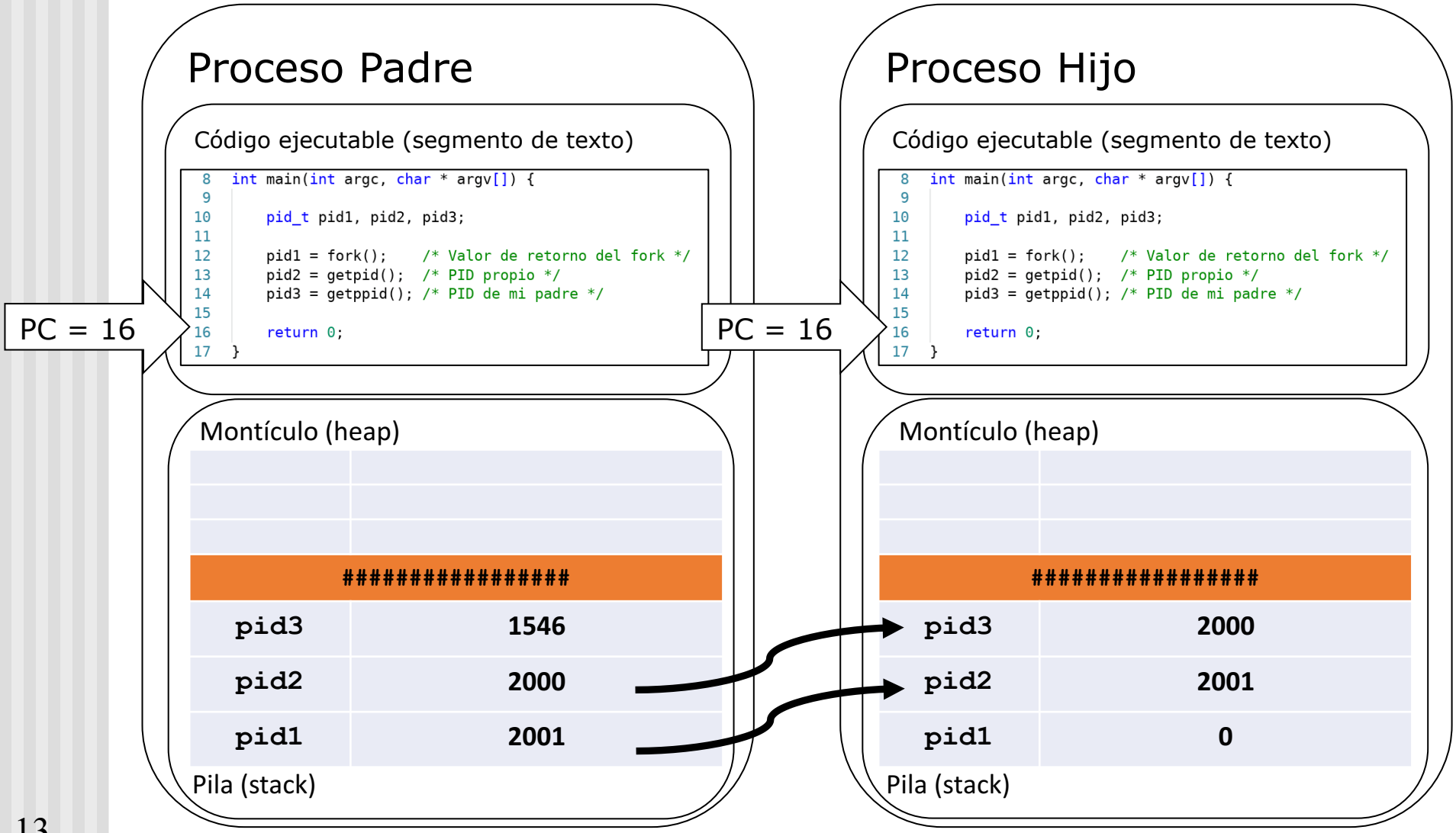
- La ejecución de ambos procesos es concurrente, y el orden en el que se intercalan las operaciones de uno u otro proceso lo determina el planificador del S.O.

Multiproceso con fork()

- PC = contador de programa

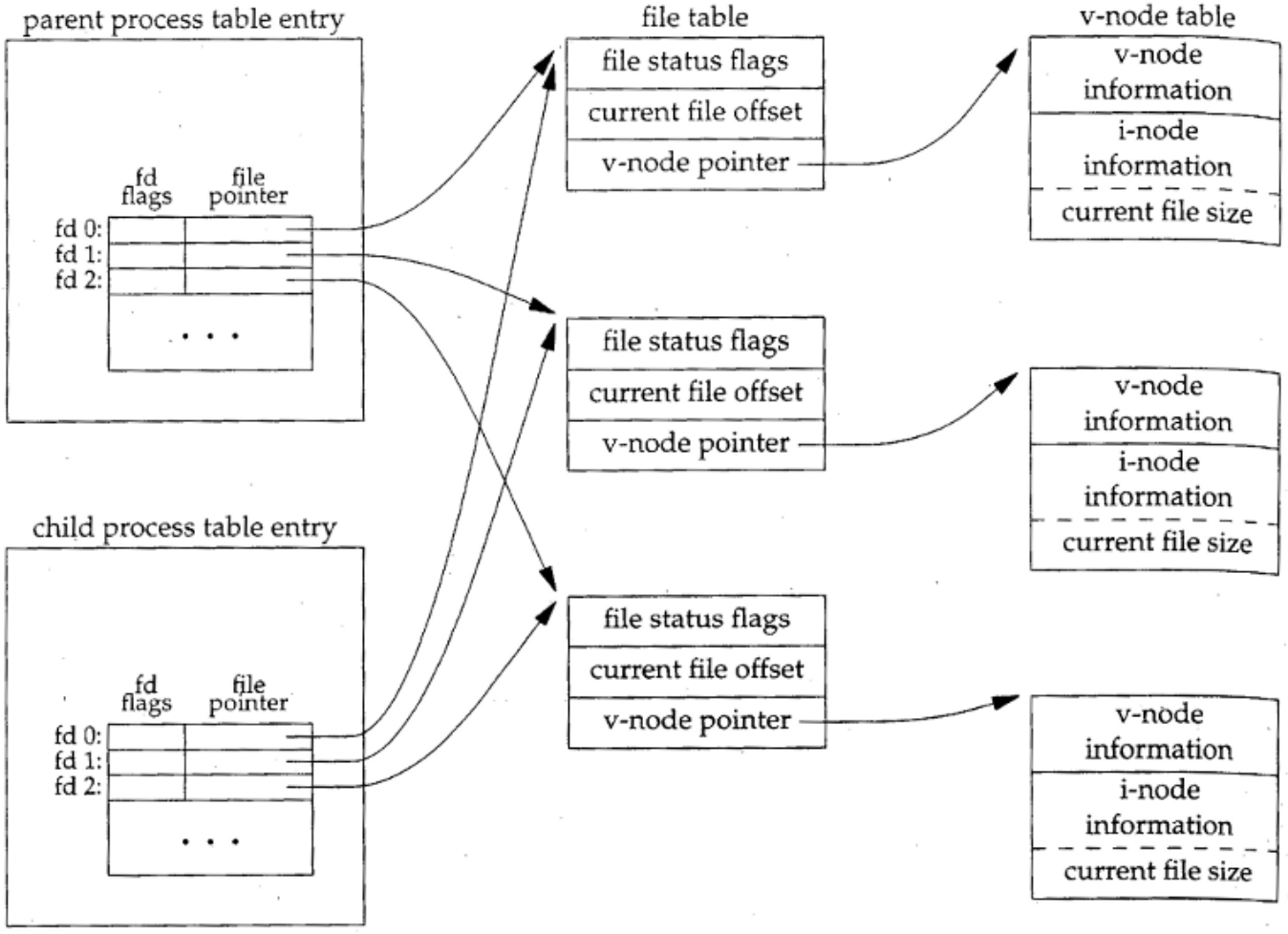


Multiproceso con fork()



Multiproceso con fork()

- Entre padres e hijos se comparten los descriptores de ficheros abiertos

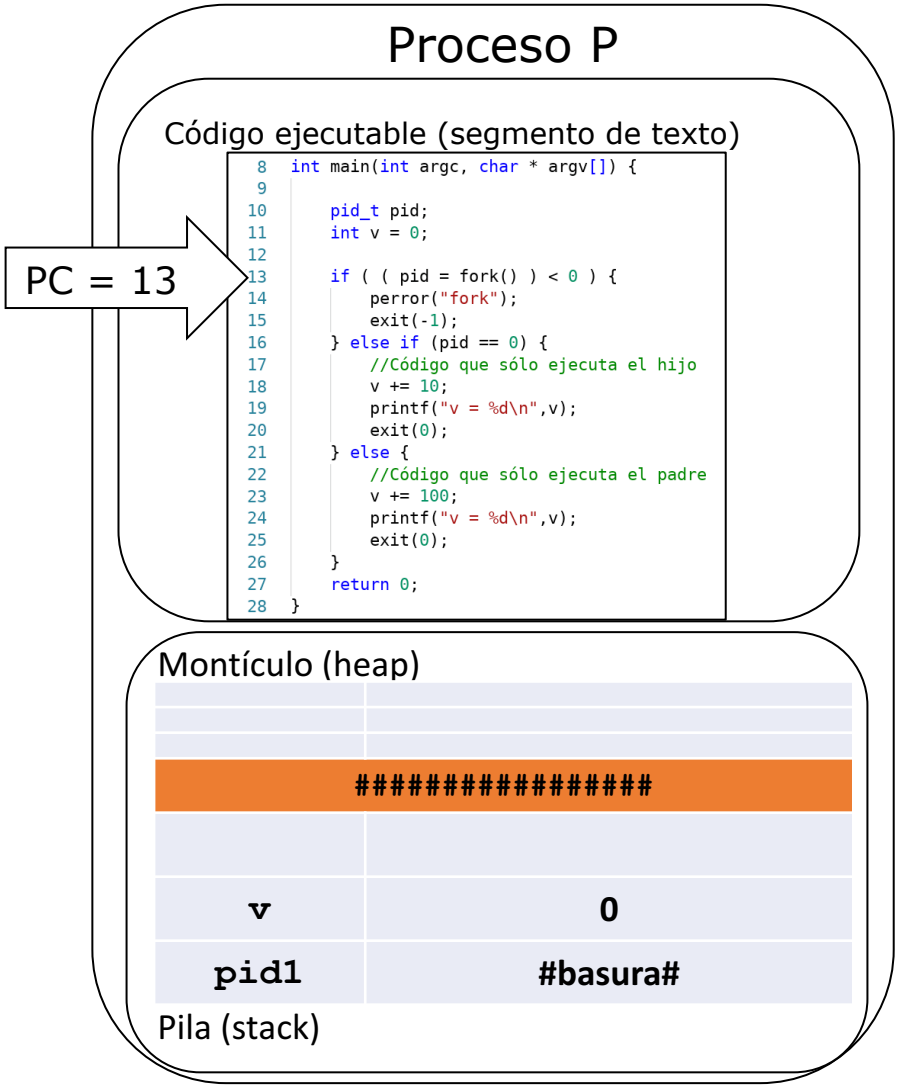


Multiproceso con fork()

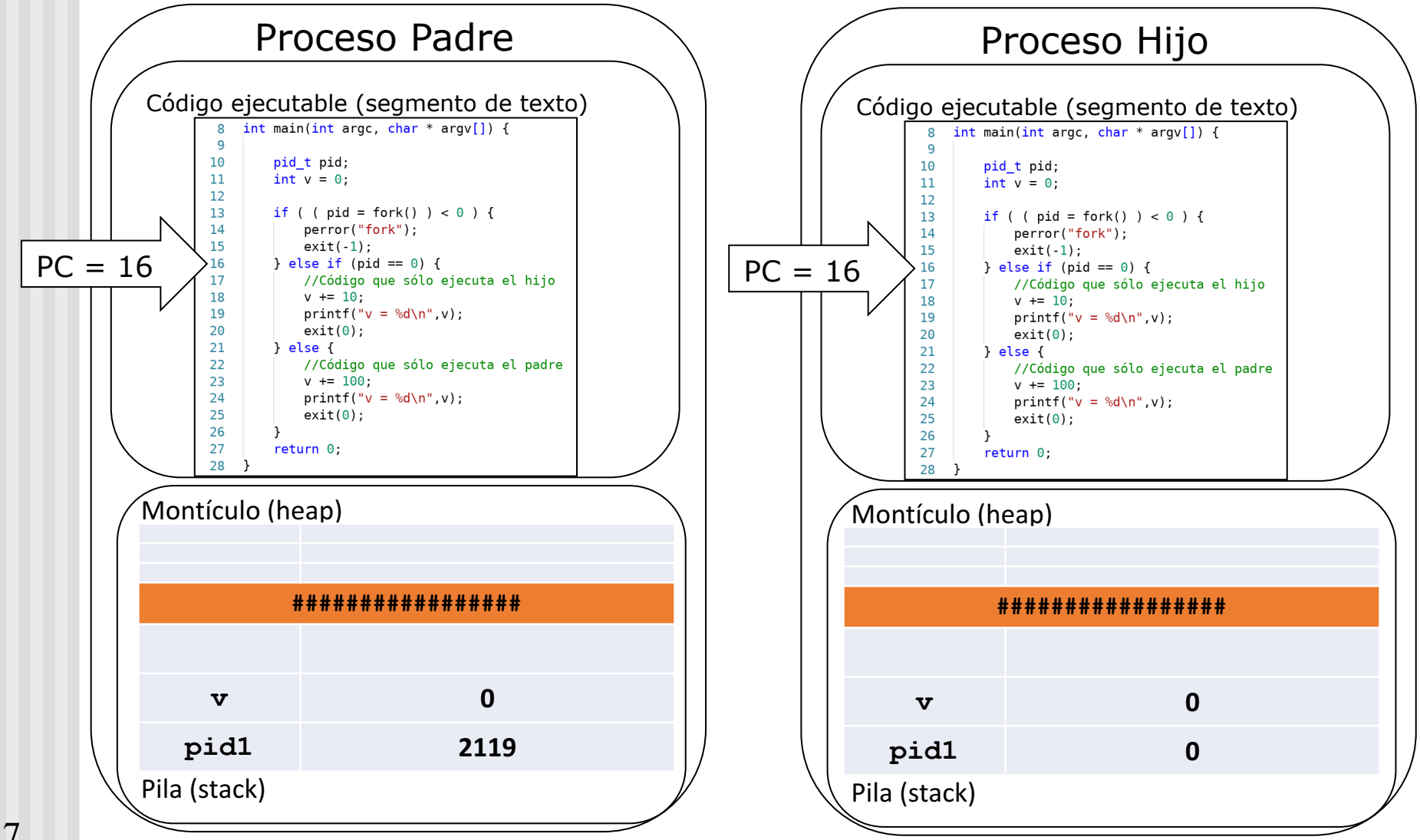
- El valor de retorno sirve para distinguir código que únicamente ha de ser ejecutado en el padre o en el hijo

```
#include <unistd.h> /* define fork() y pid_t. */
#include <sys/wait.h> /* define wait() */
int main(){
        pid_t pid_hijo;
        if( (pid_hijo = fork()) == -1){
                perror("No hay recursos!");
                exit(1);
        } else if (pid_hijo == 0) { /* Codigo del hijo */
                printf("Soy un hijo y debo terminar con exit SIEMPRE!!");
                exit(0);
        } else {  /* Codigo del padre */
                printf("Soy un mal padre: no espero a mi hijo");
                exit(0);
        } /* else */
        return 0;
} /* main */
```

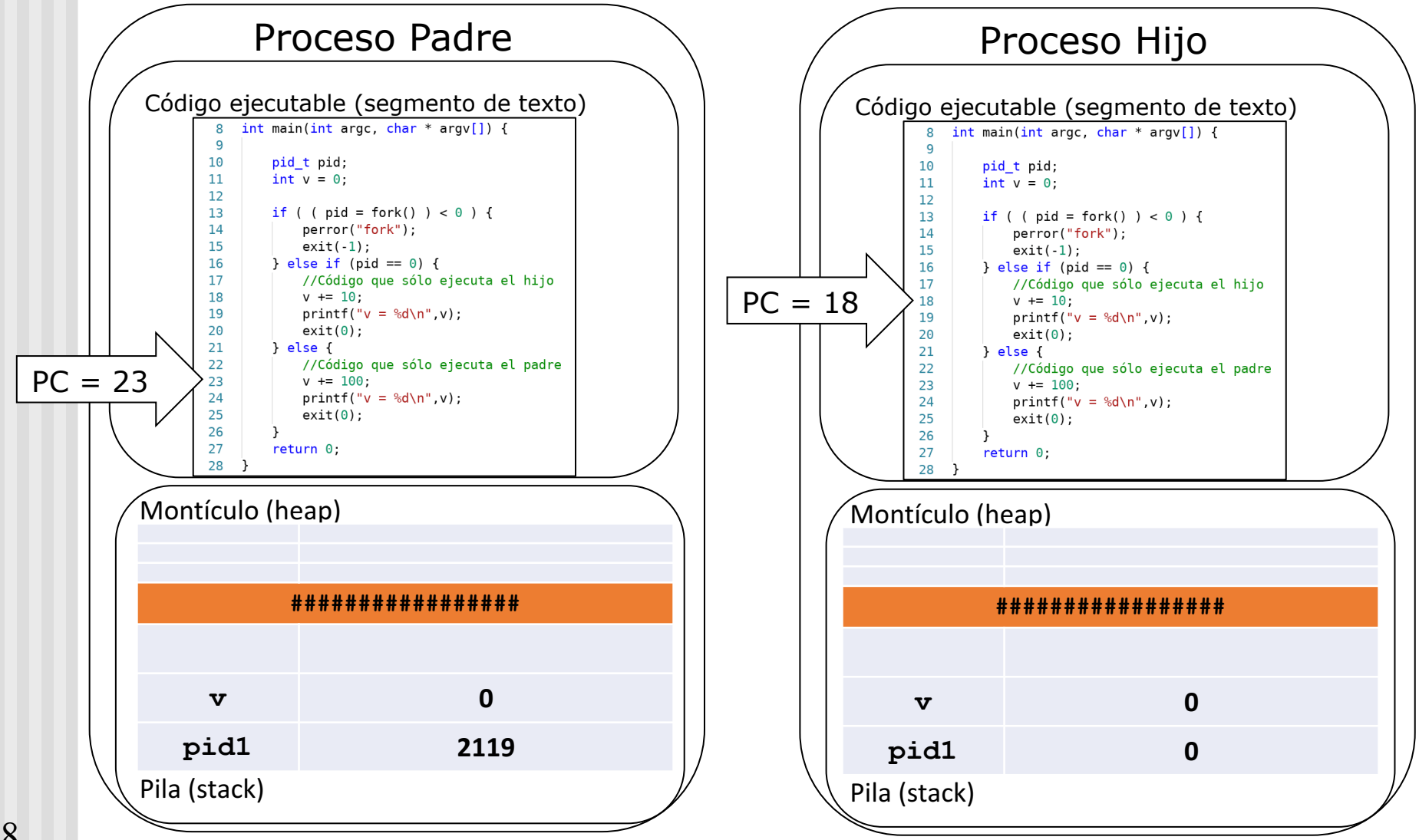
fork(): distinguir entre padre e hijo



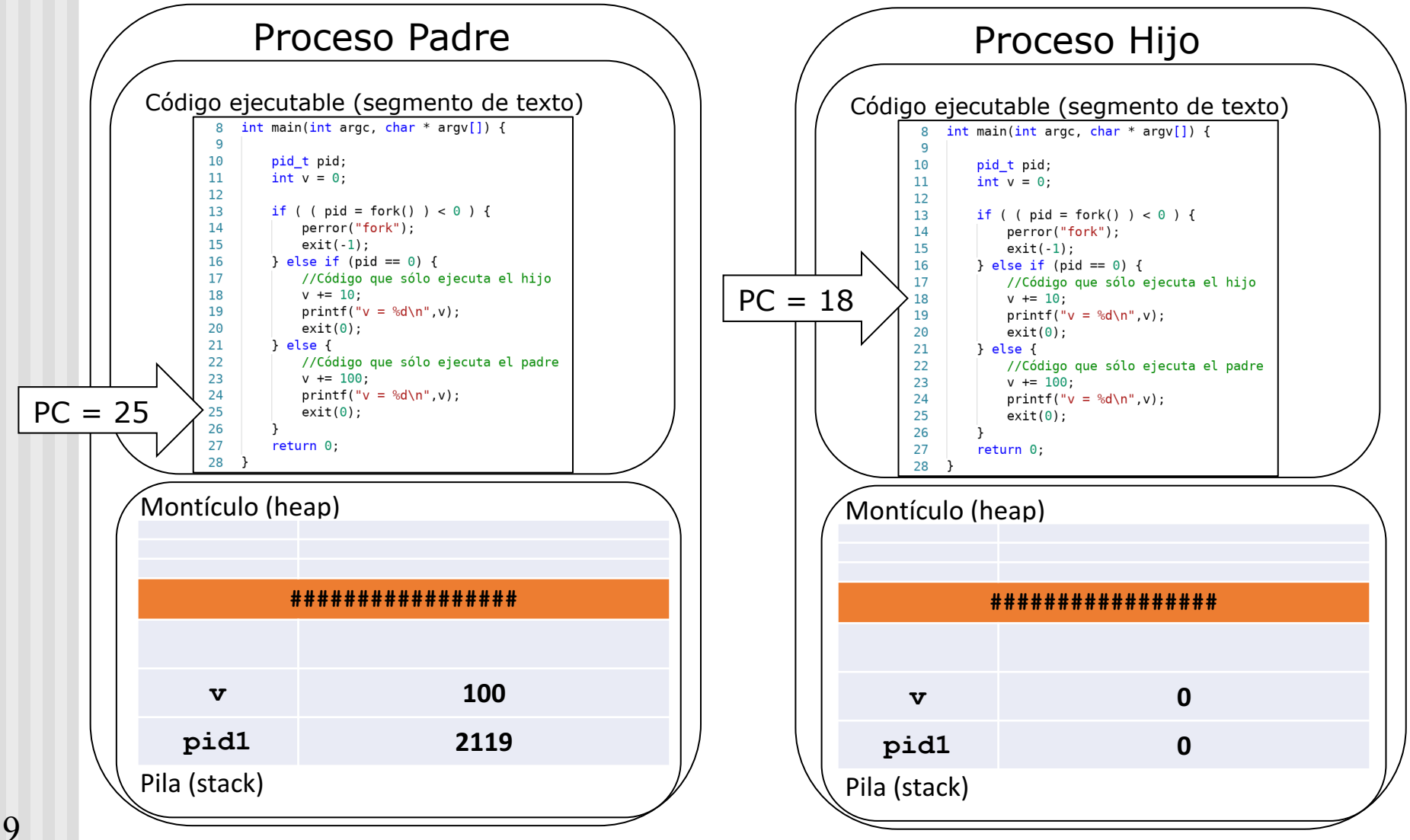
Inicio de la ejecución concurrente



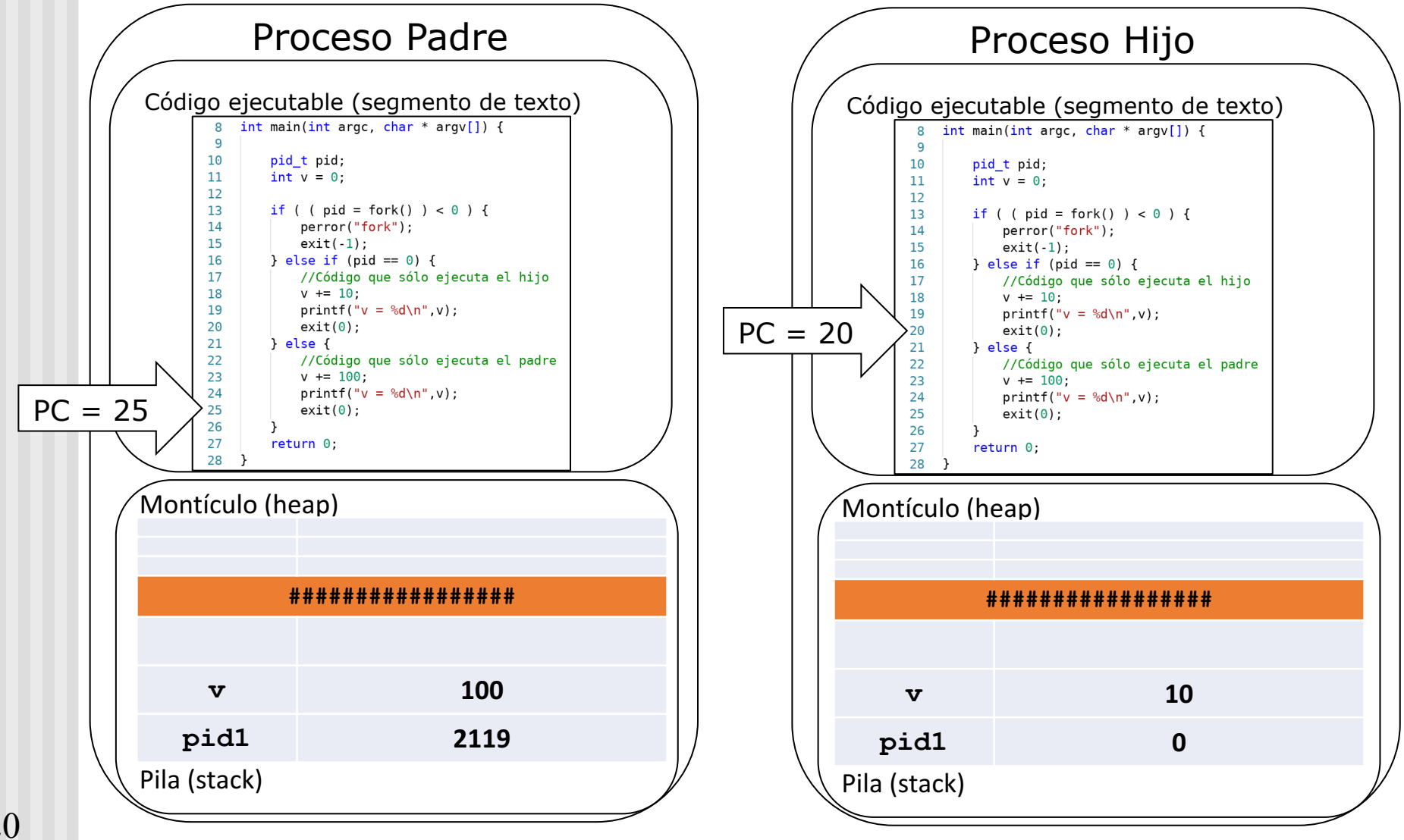
Posible intercalado de operaciones: depende del planificador del S.O.



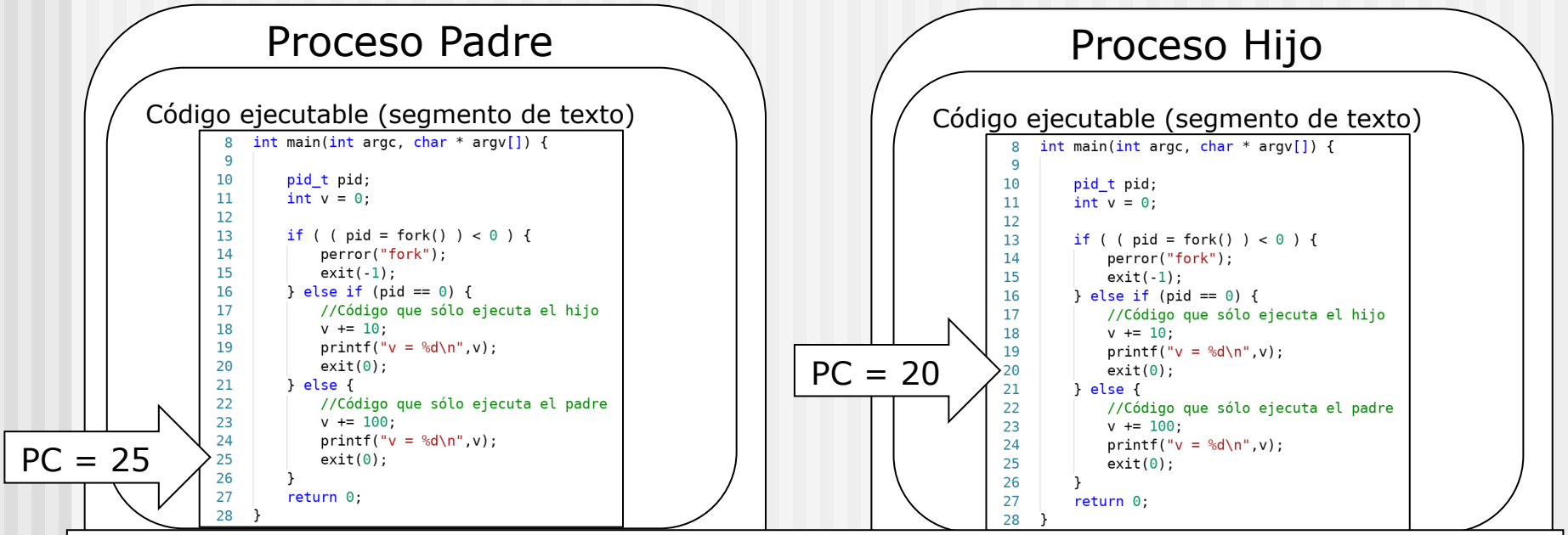
Posible intercalado de operaciones: depende del planificador del S.O.



Posible intercalado de operaciones: depende del planificador del S.O.



Posible intercalado de operaciones: depende del planificador del S.O.




```
 8  int main(int argc, char * argv[]) {
 9
10      pid_t pid;
11      int v = 0;
12
13      if ( ( pid = fork() ) < 0 ) {
14          perror("fork");
15          exit(-1);
16      } else if (pid == 0) {
17          //Código que sólo ejecuta el hijo
18          v += 10;
19          printf("v = %d\n",v);
20          exit(0);
21      } else {
22          //Código que sólo ejecuta el padre
23          v += 100;
24          printf("v = %d\n",v);
25          exit(0);
26      }
27      return 0;
28  }
```

Resultado de la ejecución:

```
alumno@debian:s10$ ./f2
v = 100
alumno@debian:s10$ v = 10
```

→ Podría haber sido diferente: depende del planificador

→ ¿Por qué aparece el símbolo del sistema?

Multiproceso con fork() (II)

- Como regla, el hijo debe terminar haciendo

  `exit(value);`

- A su vez, el padre debe esperar la finalización de su hijo ya que, si no lo hace, el hijo queda como proceso ZOMBIE en el sistema

- Se usa la función

  `wait(0);`

- Es una función bloqueante: el proceso padre espera hasta que el hijo termine

```
...
        else {  /* Codigo del padre */
                printf("Soy un buen padre: espero a mi hijo");
                wait(0);
        } /* else */
```

Procesos Zombie

- A veces un padre está interesado en el estado en el que han terminado sus hijos... y otras no
  - Para que el S.O. libere los recursos asociados en kernel a un hijo, el padre debe esperarlo
- Mientras el sistema operativo no detecte que el padre hace un wait(), convierte al hijo en zombie y no libera sus recursos
  - Esto hace que continúe en el sistema mientras el padre se está ejecutando
  - Los procesos zombies aparecen en la tabla de procesos (ps aux) como <defunct> (difuntos)

Procesos Zombie (II)

- El único problema de llamar a wait() es que es bloqueante
  - Bloquea la ejecución del padre

- Sin embargo, hay una manera de tratar la finalización de un hijo de forma asíncrona en el padre:
  - El S.O. genera la señal SIGCLD (o SIGCHLD) cada vez que un hijo termina
  - La señal se puede capturar con un manejador de señal y dentro hacer la llamada a wait()

Procesos Zombie (III)

- En Unix System V se permite una forma para indicar al S.O que no genere zombies

signal(SIGCLD, SIG_IGN);

- Para BSD y compatibilidad en general se prefiere el uso de manejadores de señal:

```
void wait_hijos(int sig){
    wait(0);
    signal(SIGCLD,wait_hijos);
} /* wait_hijos */

main(){
...
if (signal(SIGCLD,wait_hijos) == SIG_ERR) {
    perror("signal");
    exit(-1);
}
...
} /* main */
```

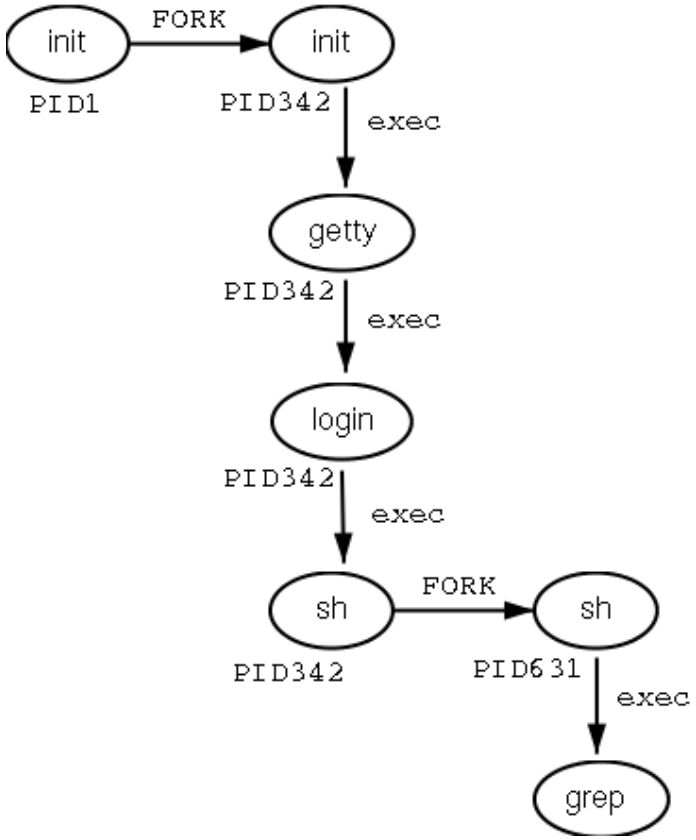
Procesos huérfanos

- Si el proceso padre acaba sin esperar a sus hijos y estos no han terminado, quedan huérfanos
  - Automáticamente los adopta el proceso init (el padre de todos, cuyo pid es 1)
  - El proceso init está programado de forma que siempre esperará a que terminen todos sus hijos antes de acabar
    - Por ejemplo, para apagar o reiniciar el S.O.

- Si el proceso padre acaba sin esperar a sus hijos, y estos ya han terminado, el proceso init los adopta e, inmediatamente, llama a wait() internamente para liberar los recursos en el sistema operativo
  - Así estos procesos desaparecen de la lista de <defunct>

La familia de llamadas exec()

- Carga una imagen de un nuevo proceso en memoria y lo ejecuta sustituyendo el espacio de direcciones previo
- Reemplaza al proceso original
- Pero mantiene el pid, ppid y los ficheros abiertos



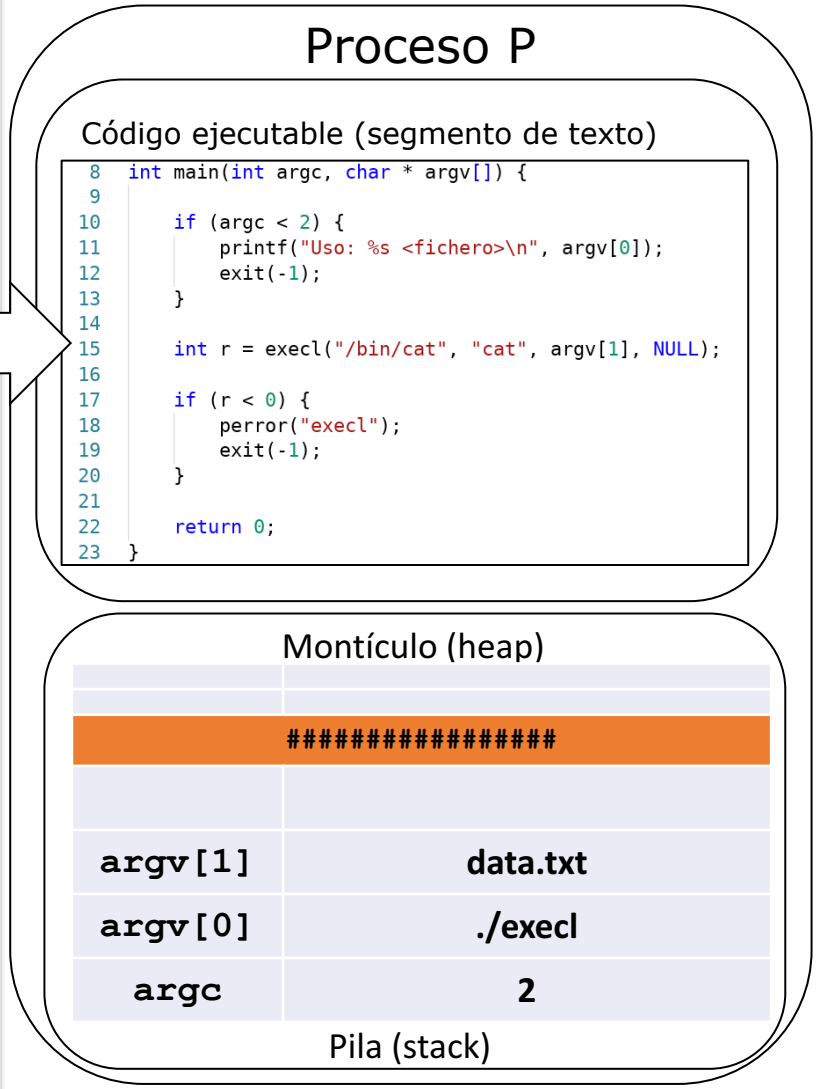

```
#include <unistd.h>
int execl ( const char *path,
            const char *arg,
            ...);
```

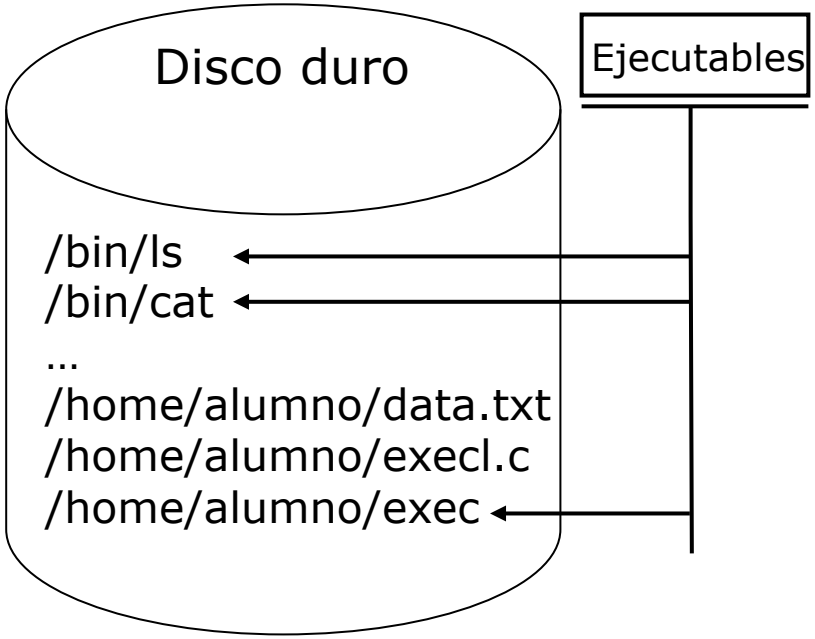
La familia de llamadas exec()

```
int ret;
ret = execl("/bin/man", "man", "2", "signal", NULL);
if (ret == -1)
   perror("execl");
```

- Primer argumento: cadena de caracteres con el ejecutable (rut absoluta en este ejemplo)
- Resto de argumentos (variables): serán los que aparezcan en l variable char *argv[] del nuevo proceso a través de su main
  - El primero es el nombre del ejecutable (argv[0])
  - Luego los argumentos del ejecutable
  - El último se pone siempre a NULL
- Si funciona execl() no devuelve nada, puesto que el proceso original es reemplazado. Si hay fallo devuelve -1

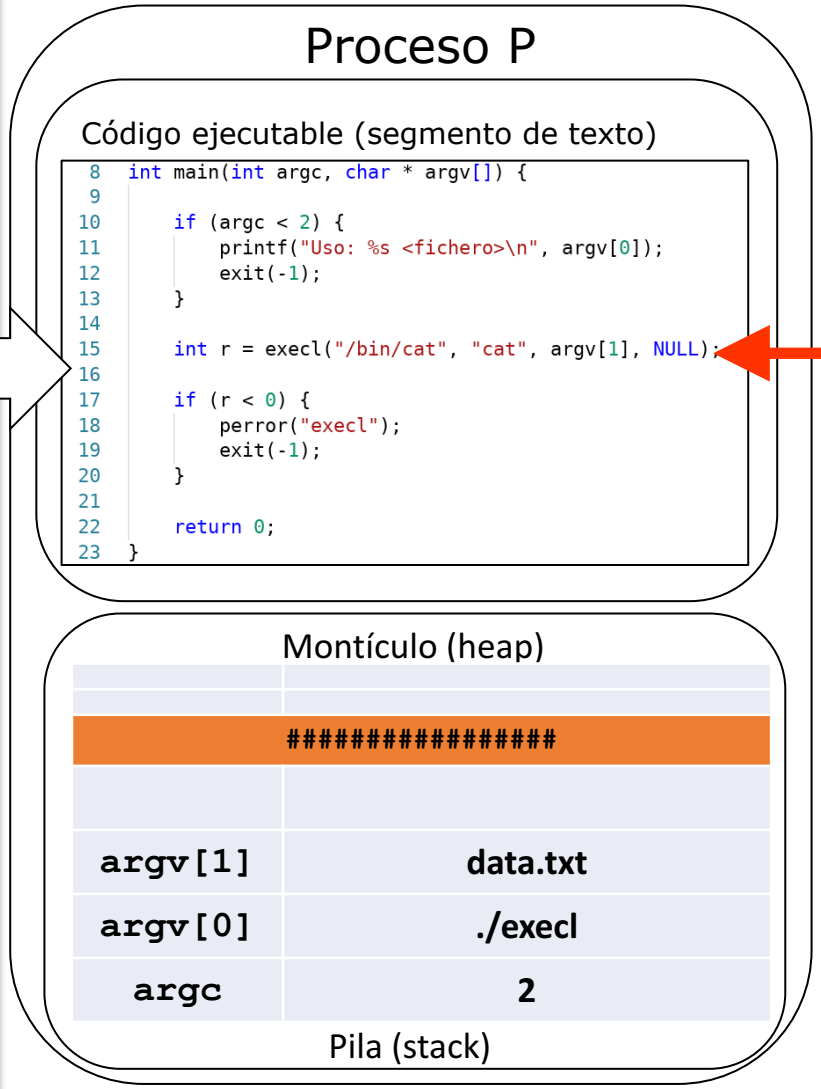
La familia de llamadas exec()

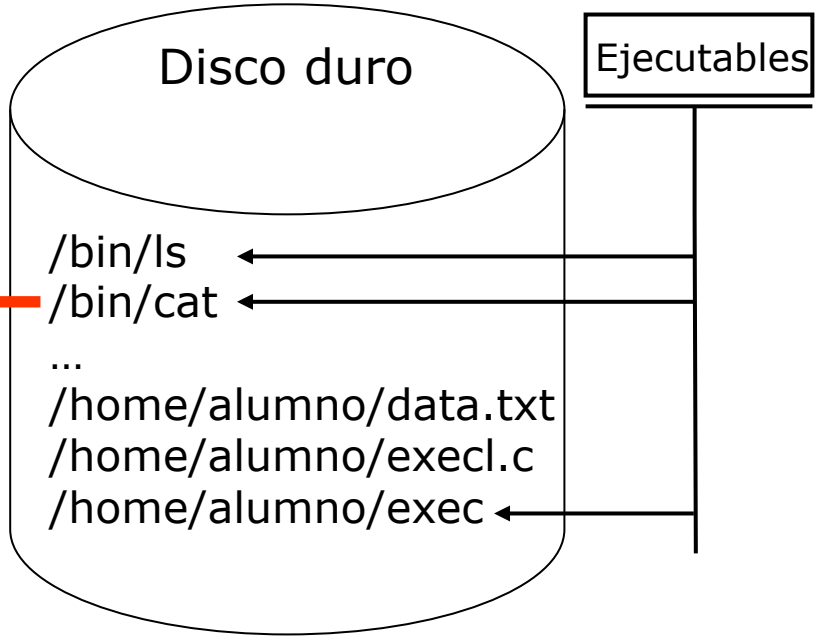




- Los ejecutables están en el disco
- execl() necesita el path completo

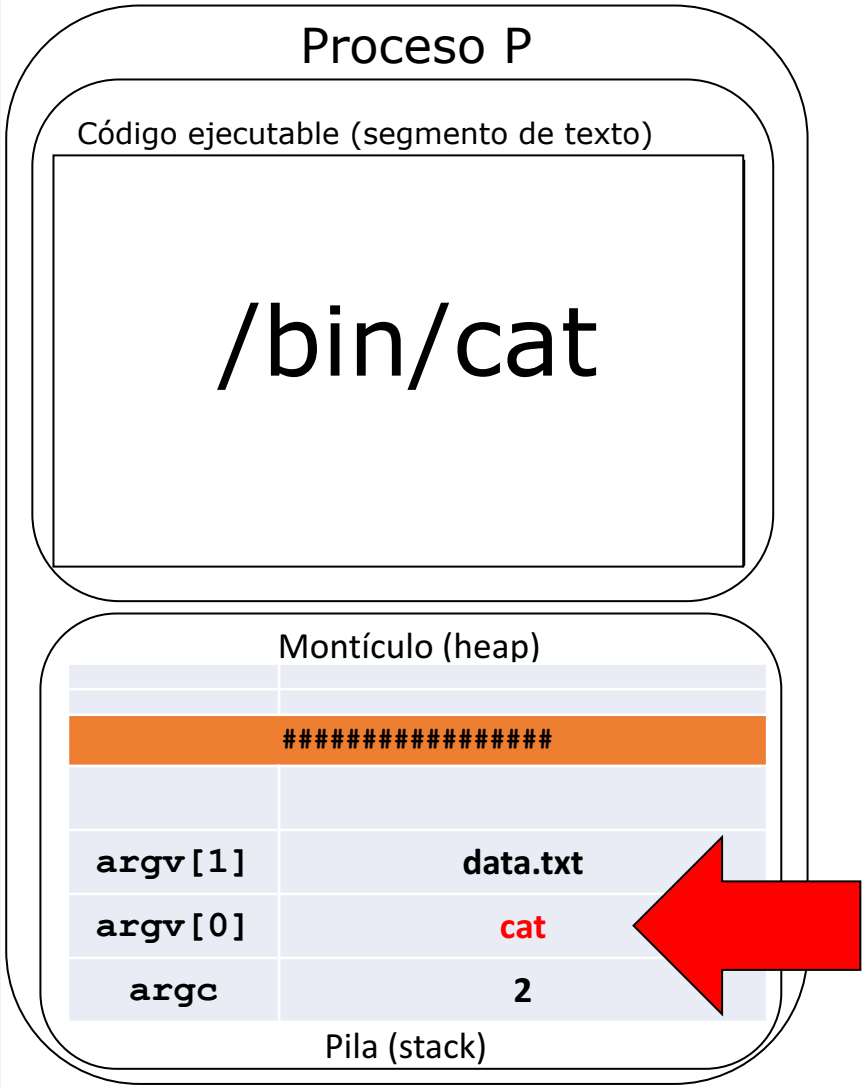
La familia de llamadas exec()

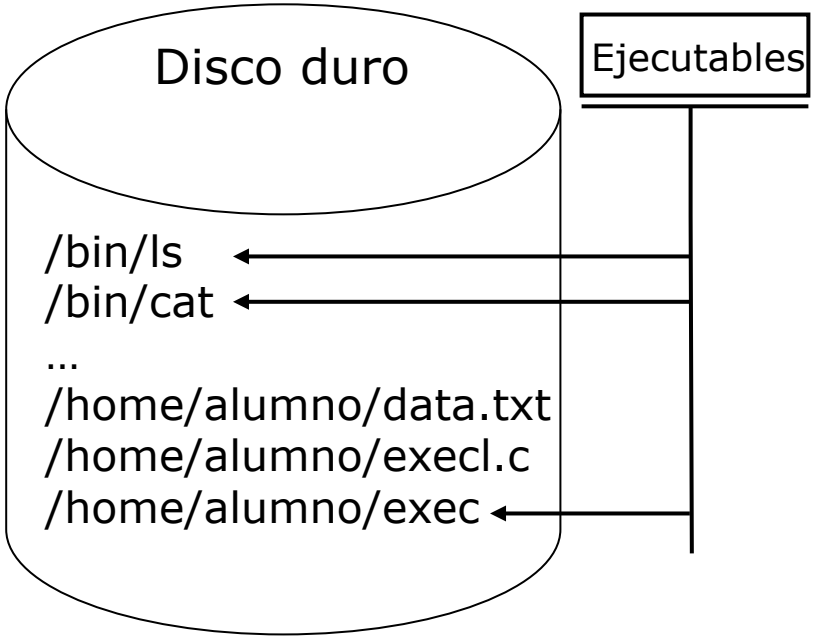




- Una vez se invoca, el código ejecutable que está en /bin/cat sobreescribe el del Proceso P
- La línea 16 sólo se ejecuta si la llamada a execl() no es correcta

La familia de llamadas exec()





- Se dice que el Proceso P ha mutado
- argv[0] en /bin/cat es ahora "cat", el segundo argumento de execl()

La familia de llamadas exec()

```
#include <unistd.h>
   int execlp(const char *file, const char *arg, ...);
   int execle(const char *path, const char *arg, ...,
              char * const
              envp[]);
   int execv (const char *path, char *const argv[]);
   int execvp(const char *file, char *const argv[]);
   int execve(const char *filename, char *const argv[],
              char *const envp[]);
```

- Si el nombre de la función tiene **l**, los argumentos del nuevo proceso se pasan como lista de parámetros
- Si el nombre de la función tiene **v**, los argumentos del nuevo proceso se pasan como array (vector)
- Si el nombre de la función tiene **p**, el nombre del proceso se busca en el PATH del usuario
- Si el nombre de la función tiene **e**, se proporcionan nuevas variables de entorno al nuevo proceso

La función system()

- Ejecuta un comando en una shell y espera de forma síncrona su resultado
  - Útil para sustituir la acción de un proceso que genera un hijo para ejecutar el comando y que inmediatamente se pone a esperarlo

```
#include <stdlib.h>
int system (const char *command);
```

- En realidad está ejecutando: **`/bin/sh -c command`**
- Durante esta ejecución la señal **SIGCHLD** se bloquea. **SIGINT** y **SIGQUIT** se ignoran
- *Ejercicio: implementa tu propia versión de **system**() usando las llamadas al sistema fork, exec y wait*