Sistemas Operativos

Gesitón de Procesos

Conceptos de Proceso e Hilo

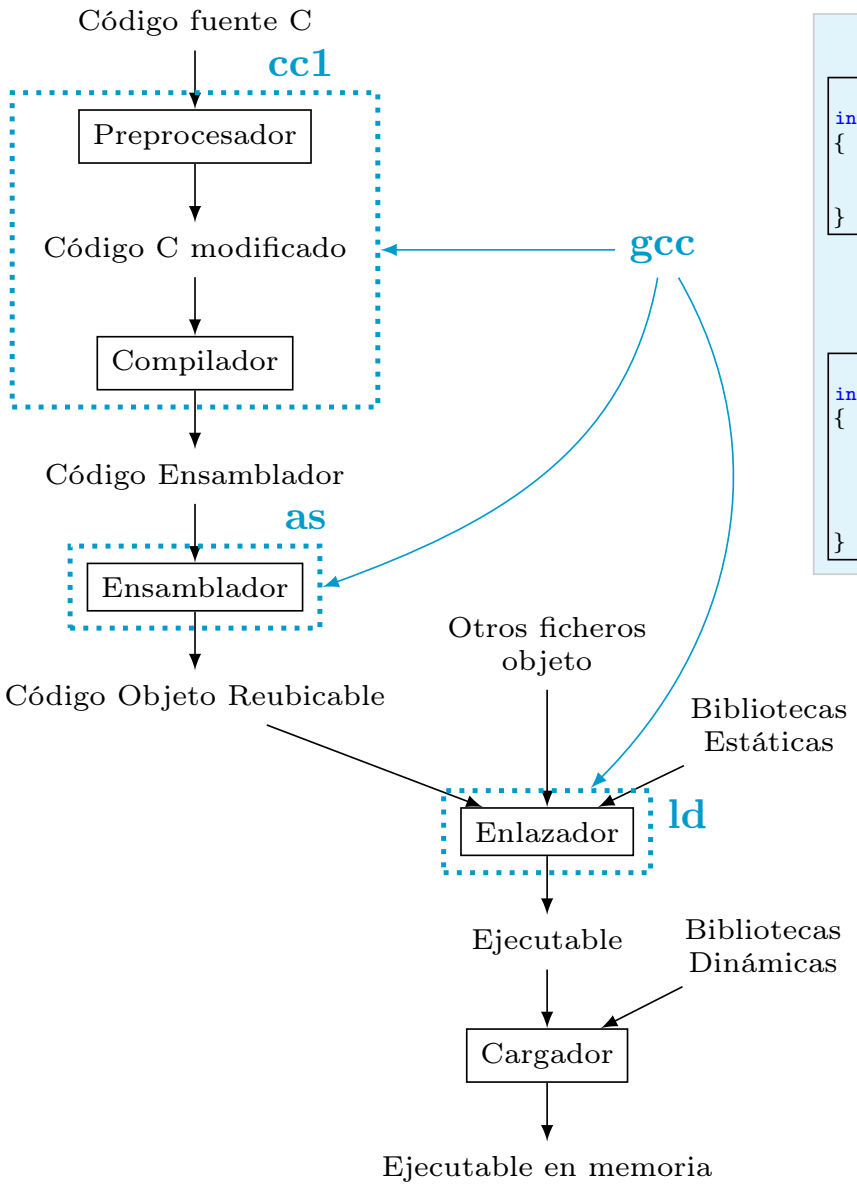
Agenda

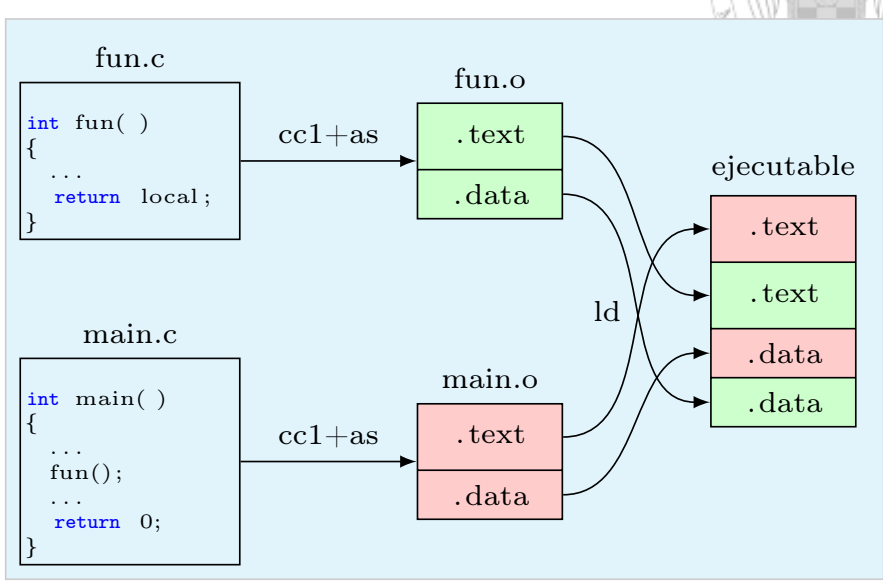
Agenda

Programa vs Proceso

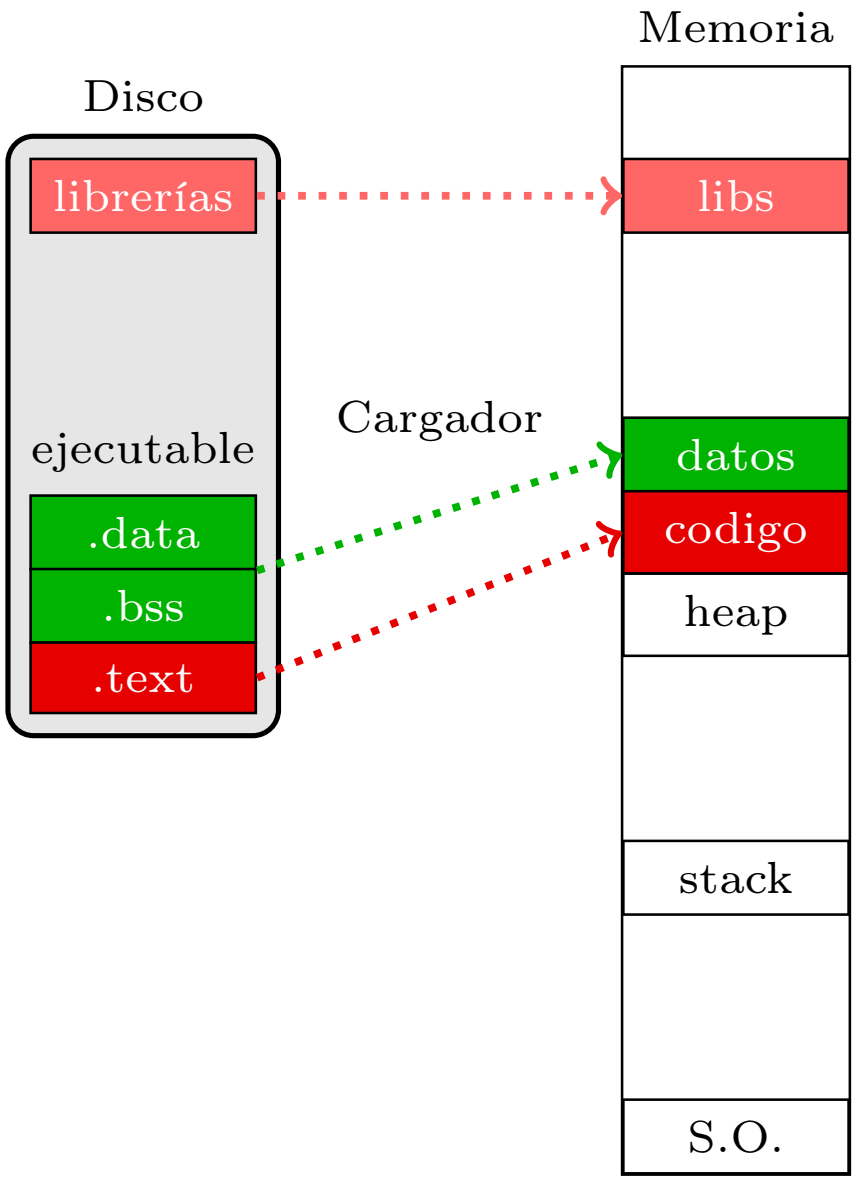
- Programa: fichero ejecutable en disco
- Proceso: una instancia de ejecución de un programa
  - Puede haber varios procesos ejecutando el mismo programa
  - El SO mantiene información/recursos asociados a procesos
    - Bloque de Control de Proceso (BCP)

Recordar: ejecutable





Proceso


Memoria
Disco
librerías
libs
Cargador
ejecutable
datos
.data
codigo
.bss
heap
.text
stack
S.O.

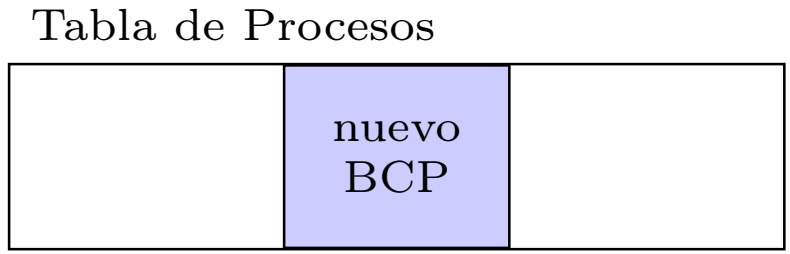


Tabla de Procesos
nuevo
BCP

Agenda

Información del proceso almacenada por el SO

La información que el SO necesita para gestionar un proceso es

- Estado del procesador
- Imagen de memoria
- Tabla de Páginas (TP) del proceso
- Información de identificación.
- Información de planificación del proceso
- Descriptores de ficheros abiertos (TFA)
- Señales recibidas (POSIX)

Bloque de Control de Procesos (BCP)

El BCP es un descriptor que mantiene el SO por cada proceso creado en el sistema. Almacena la información necesaria para gestionar el proceso.

Estado del procesador

- Formado por el contenido de los registros arquitectónicos:
  - Registros generales
  - Contador de programa
  - Puntero de pila
  - Registro de estado
  - Registros especiales
- Cuando un proceso está ejecutando su estado reside en los registros del computador.
- Cuando un proceso no ejecuta su estado reside en el BCP.

Imagen de Memoria (1/2)

El espacio virtual de direcciones de un proceso se divide en regiones o segmentos:

- Código
- Datos con valor inicial
- Datos sin valor inicial
- Heap
- Pila

Esta distribución es lógica/virtual, no se corresponde con su almacenamiento real en memoria física (RAM)

- Las direcciones que maneja un programa son direcciones virtuales

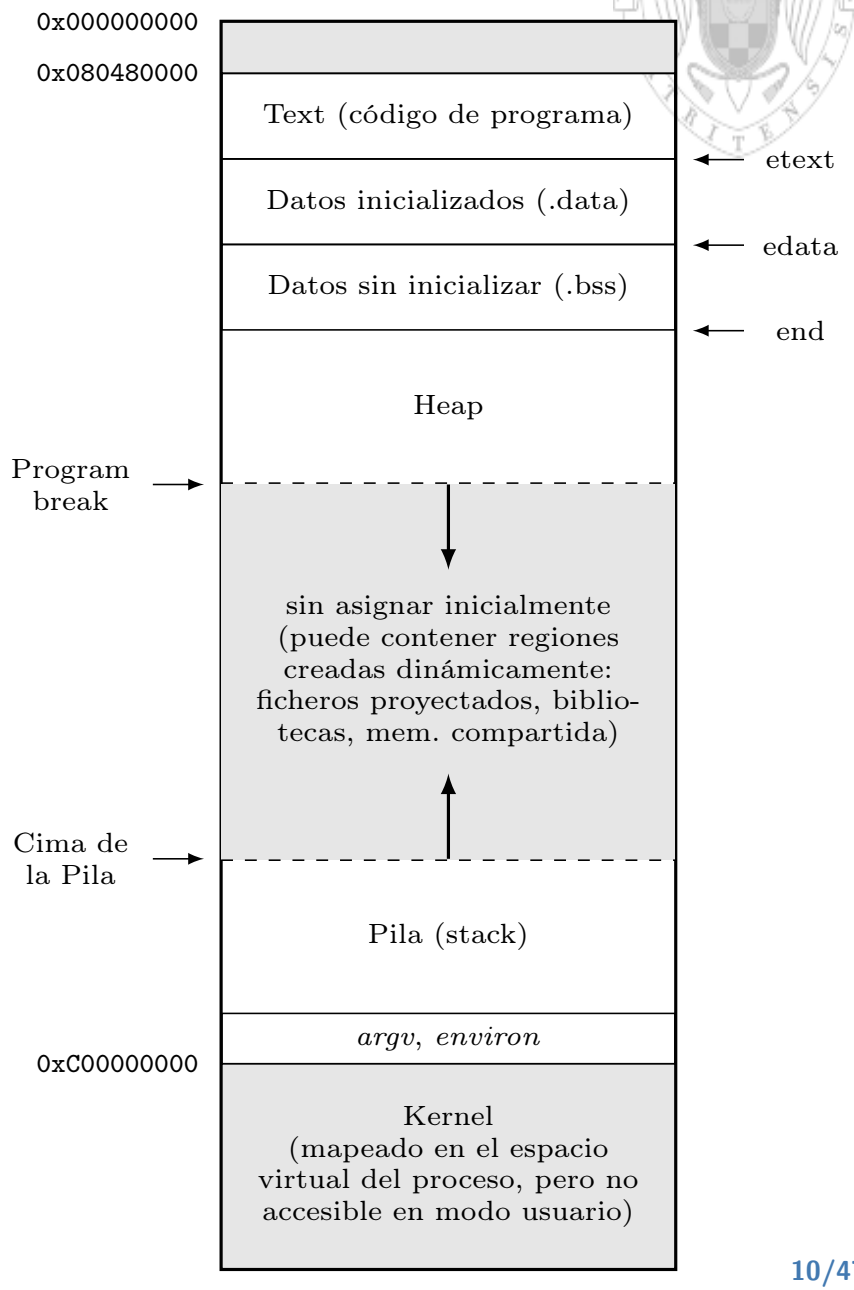


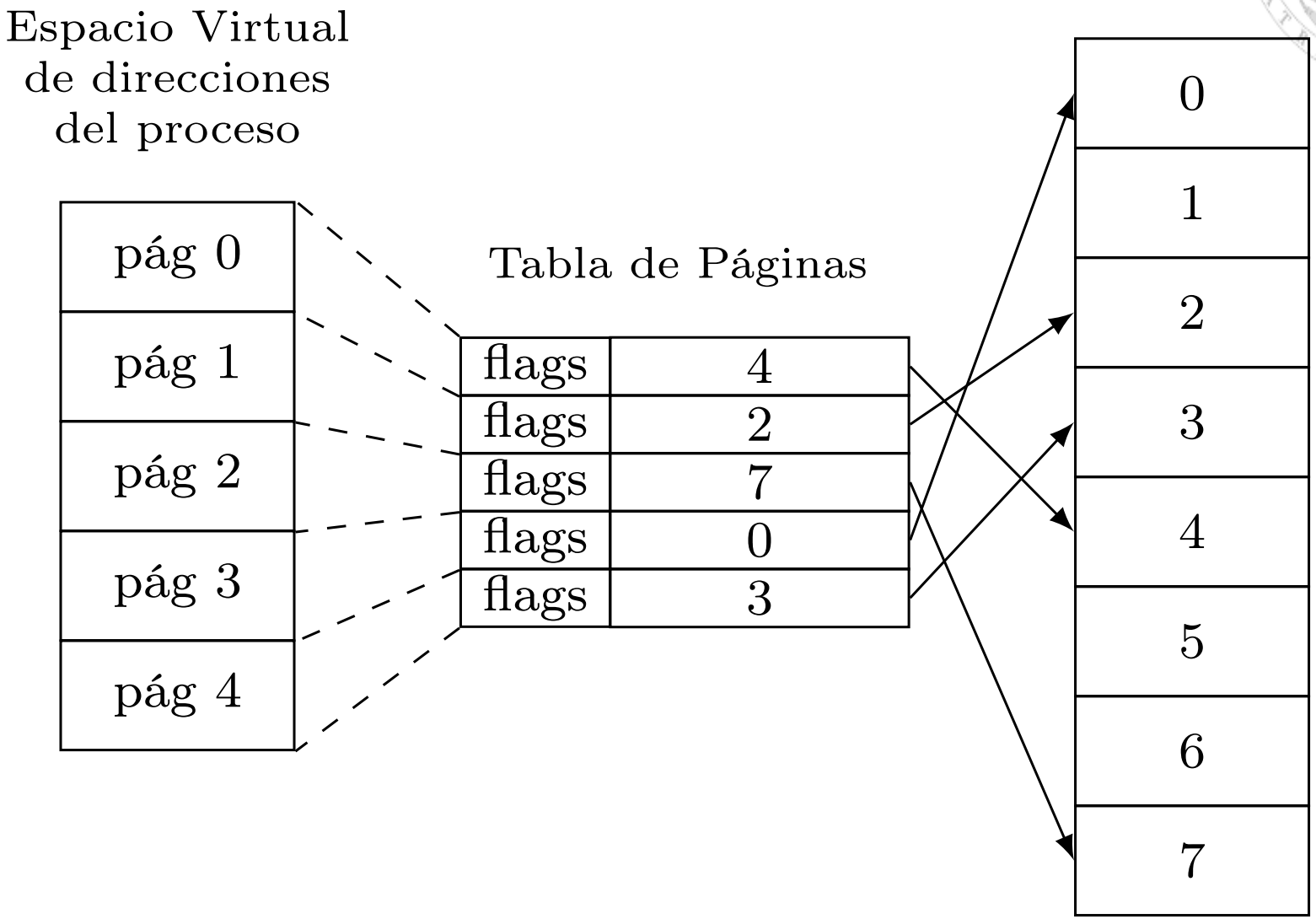
Imagen de Memoria (2/2)

En Linux podemos consultar las regiones válidas de la imagen de memoria de un proceso activo:

- a través del fichero /proc/pid/maps, donde pid sería el pid del proceso.
- mediante el comando pmap.

Consultar `man proc` y `man pmap`.

Tabla de páginas



Permite la traducción de direcciones virtuales a direcciones físicas

Información de identificación

En los sistemas POSIX es habitual encontrar la siguiente información de identificación:

- del proceso y proceso padre (pid, ppid)
- del usuario y grupo (uid/gid)
- del usuario y grupo efectivos (euid y egid)

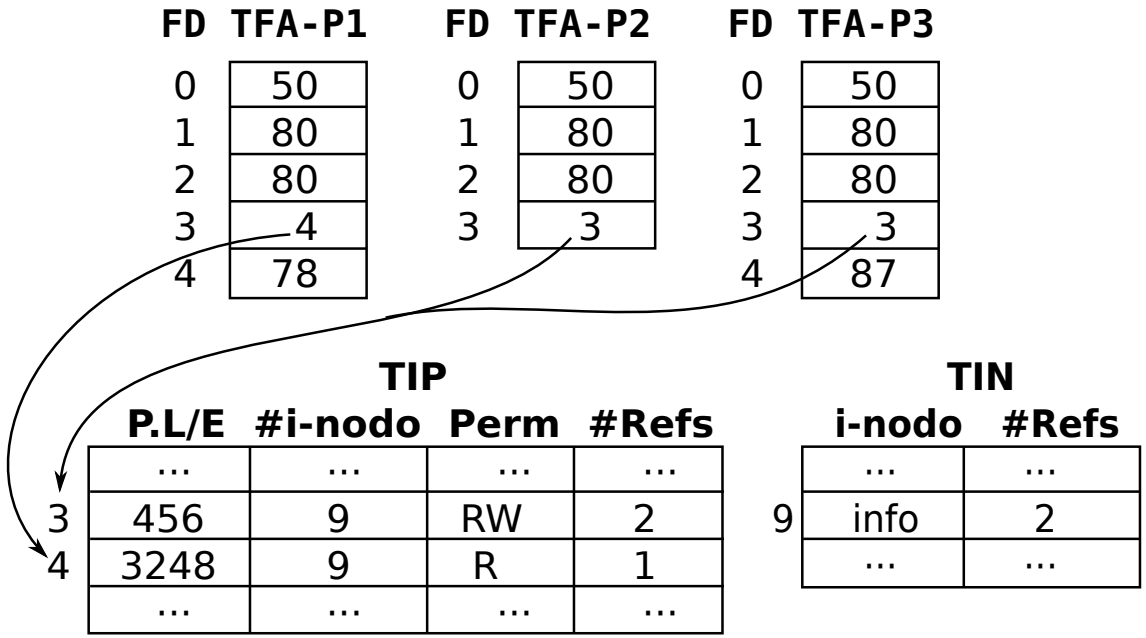
Información de planificación del proceso

En realidad depende de la estrategia de planificación escogida, pero suele contener:

- Estado del proceso
- Tiempo de cpu
- Prioridad (nice)
- Tipo de proceso (tiempo real, ...)

Tabla de descriptores abiertos (TFA)

- Una tabla con una entrada por descriptor de fichero abierto (TFA)
- Cada entrada contiene una referencia a la entrada de la Tabla Intermedia de Posiciones (TIP) con la información de la apertura:
  - El puntero de posición, los permisos de la apertura, nº de procesos que usan la entrada, etc.
- POSIX: Cuando un proceso crea otro (`fork()`) la TFA se copia del proceso padre al hijo, incrementando el nº de referencias en la TIP:
  - Comparten el puntero de L/E



Señales recibidas (POSIX)

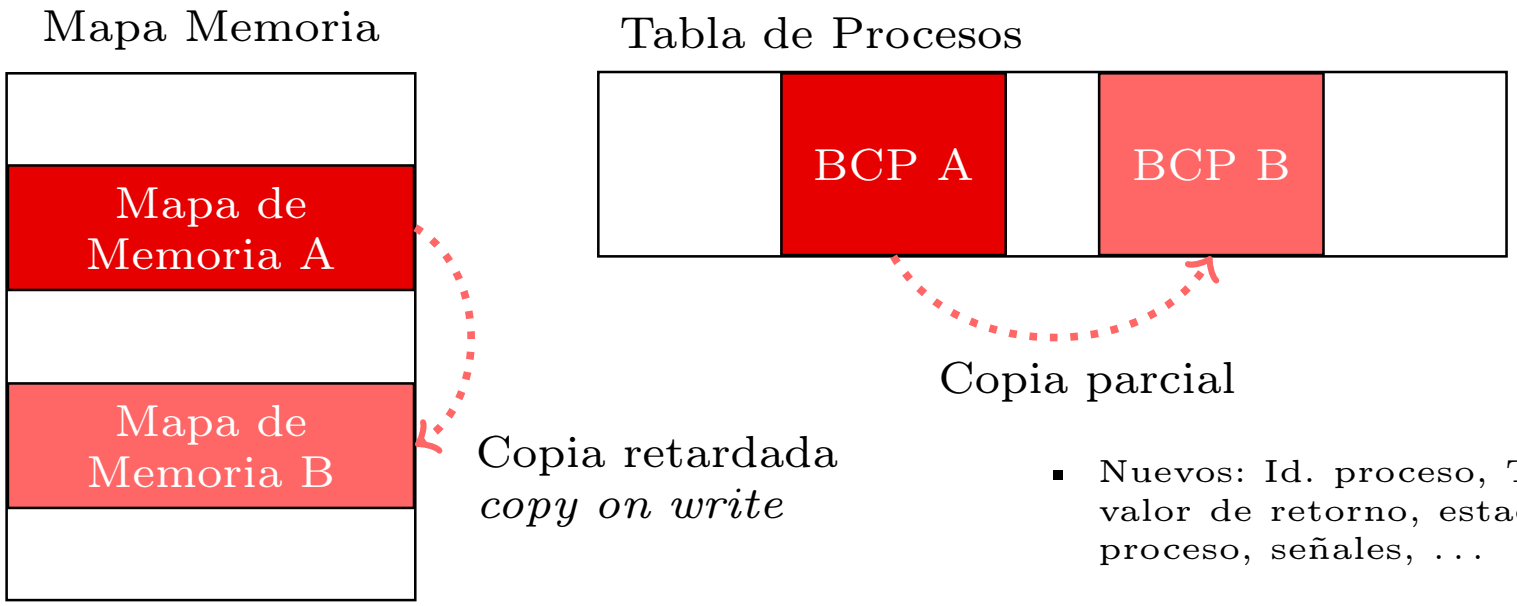
- En POSIX los procesos pueden recibir señales
  - de otros procesos: `int kill(pid_t pid, int sig)`
  - del propio SO
- El SO guarda una máscara de señales pendientes por atender para cada proceso (no encola varias señales del mismo tipo)
- Un proceso puede:
  - bloquearse a la espera de la recepción de alguna señal:
    `int pause(void)`
  - registrar un manejador para tratar la señal:
    `int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);`
- Cuando el SO cede la CPU al proceso, si tiene señales pendientes serán tratadas primero.
  - Supone una analogía software de una interrupción

Agenda

Creación de proceso (clonado)

Llamada al sistema `pid_t fork(void)`

- Clona el proceso que la ejecuta
- Noción de procesos padre (el original) e hijo (el nuevo)
- El hijo tiene accesos a recursos del padre (copia parcial de BCP)
- Valor de retorno distinto para padre (PID del hijo) e hijo (0)
- El proceso hijo sólo tiene un hilo.



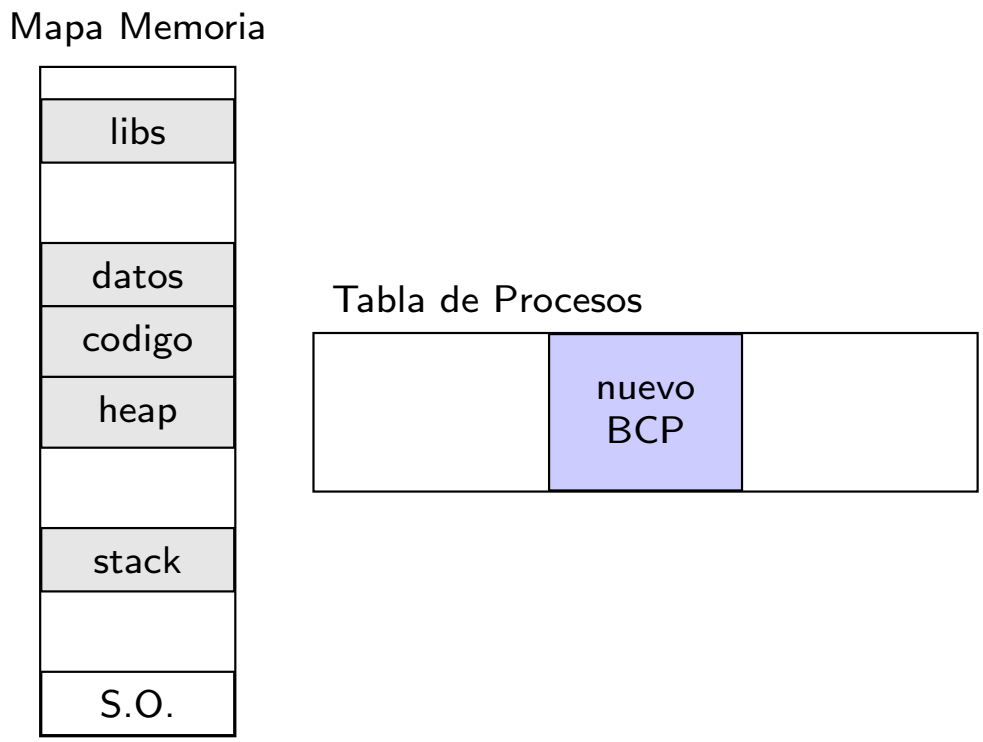
- Nuevos: Id. proceso, TP, valor de retorno, estado del proceso, señales, ...
- Copiados: TFA, Id. usuario, resto del estado arquitectónico, prioridad, ...

Ejemplo de uso de fork

```
void main()
{
    pid_t pid;
    ...
    pid = fork();
    if (pid == 0) {
        /* proceso hijo */
        ...
    }
    else if (pid > 0){
        /* proceso padre */
        ...
    }
    else{
        /* proceso padre, tratamiento de error */
        ...
    }
    ...
}
```

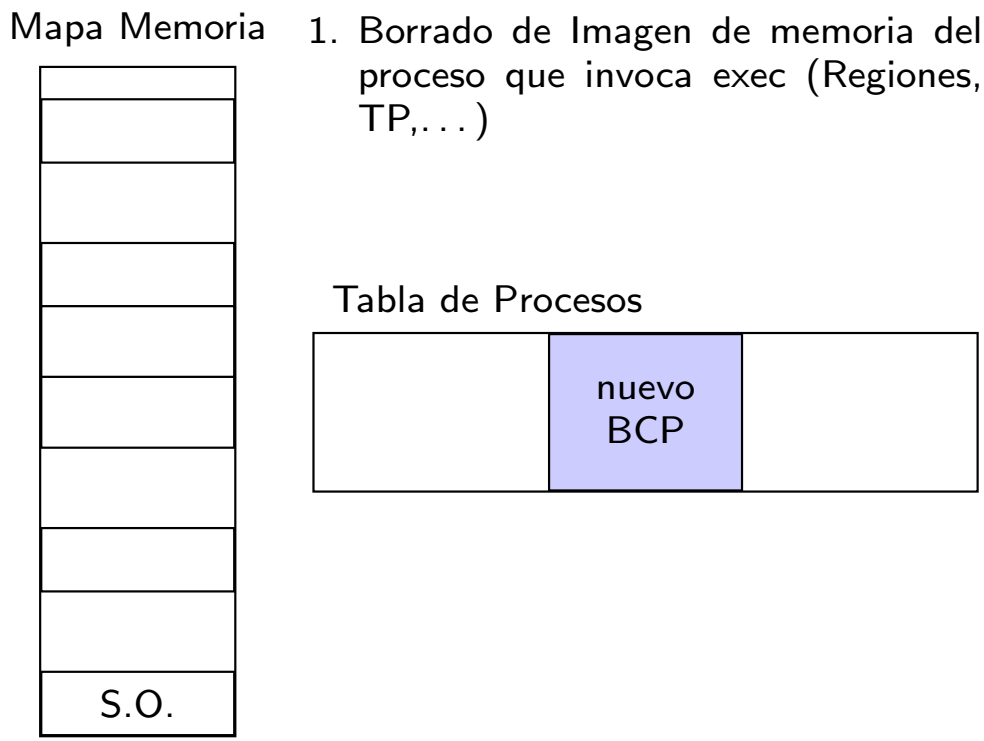
Cambio de programa - exec

La familia de llamadas al sistema `int exec*(const char *path, ...)` permite cambiar el mapa de memoria del proceso por el de un nuevo programa con argumentos de llamada



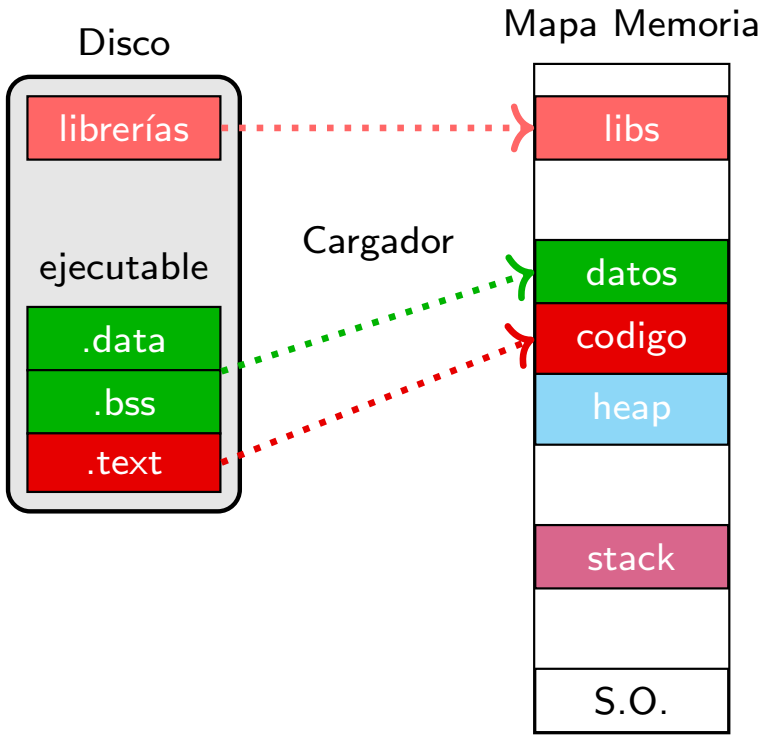
Cambio de programa - exec

La familia de llamadas al sistema `int exec*(const char *path, ...)` permite cambiar el mapa de memoria del proceso por el de un nuevo programa con argumentos de llamada



Cambio de programa - exec

La familia de llamadas al sistema `int exec*(const char *path, ...)` permite cambiar el mapa de memoria del proceso por el de un nuevo programa con argumentos de llamada



1. Borrado de Imagen de memoria del proceso que invoca exec (Regiones, TP,...)
2. Carga del nuevo mapa de memoria

Tabla de Procesos

exec: ficheros y señales

Ficheros

El proceso mantiene la TFA, por tanto el nuevo programa tiene acceso a los ficheros abiertos por el anterior, a menos que se hubiesen abierto con la opción *close on exec* (flag `O_CLOEXEC` activo)

exec: ficheros y señales

Ficheros

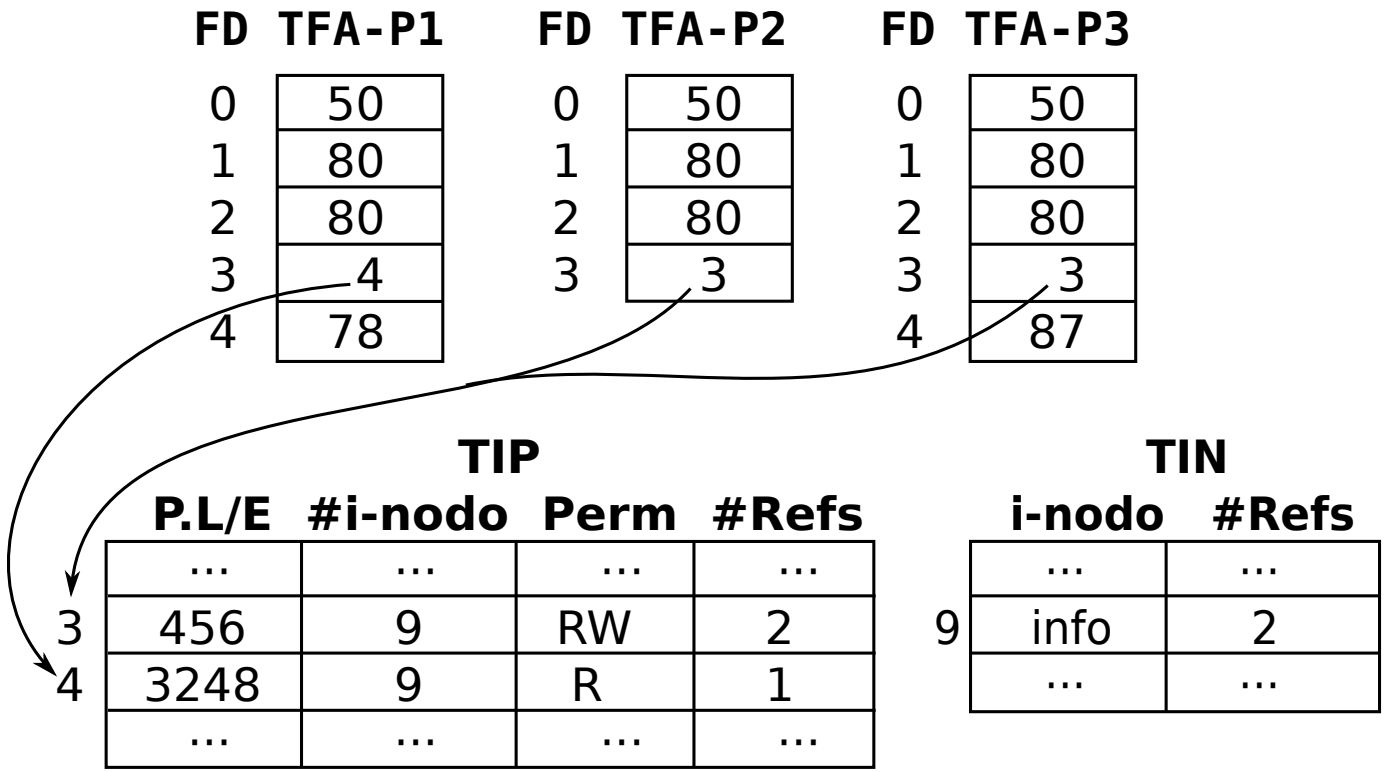
El proceso mantiene la TFA, por tanto el nuevo programa tiene acceso a los ficheros abiertos por el anterior, a menos que se hubiesen abierto con la opción *close on exec* (flag `O_CLOEXEC` activo)

Señales

Las señales que tengan registrado un manejador pasarán a tener la acción por defecto (generalmente matar el proceso)

Recordar: tablas del servidor de ficheros

- Locales, una por proceso:
  - Tabla de descriptores de ficheros abiertos (TFA)
- Globales, una para todo el sistema:
  - Tabla intermedia de posiciones (TIP).
  - Tabla intermedia de nodos-i (TIN).



Procesos y ficheros: semántica POSIX

- Distintos procesos pueden tener abierto el mismo fichero con distintas aperturas
  - El sistema manitene coherentes los atributos, manteniendo por separado la tabla de nodos-i
  - Las escrituras a fichero son inmediatamente visibles para todos los procesos con dicho fichero abierto
    - Es responsabilidad del programador sincronizar los procesos
- Además, los procesos pueden compartir aperturas de fichero:
  - En fork se copia la tabla de ficheros abiertos
  - En exec sólo se cierran los descriptores con flag `O_CLOEXEC`
  - Las entradas de las tablas de TFA apuntan a la entrada de la tabla TIP, creada en la apertura
    - Contiene el marcador de posición
  - Usan entonces el mismo marcador de posición en el fichero
    - Una operación sobre el fichero en un proceso afecta a la posición de la siguiente operación en otro proceso
  - Mecanismo utilizado por el shell para redireccionar la entrada y la salida estándar a ficheros o a pipes
    - Usando las llamadas al sistema `dup2` y `pipe`

Variantes de exec

Variantes:

- Formato largo (l): un puntero por argumento
- Formato vector (v): un array con los parámetros
- Con entorno (e)
- Buscando en la variable PATH (p)

Cabeceras:

- `int execl(const char *path, const char *arg0, ... , NULL)`
- `int execle(const char *path, const char *arg0, ... , NULL, char *const envp[])`
- `int execlp(const char *file, const char *arg0, ... , NULL)`
- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`
- `int execvpe(const char *file, char *const argv[] char *const envp[])`

Esperar la terminación de hijos

Un proceso puede bloquearse hasta que finalice alguno de sus hijos:

- `pid_t wait(int *status)`
  espera la finalización de cualquiera de sus procesos hijo
- `pid_t waitpid(pid_t pid, int *status, int options)`
  - `pid` determina por qué proceso se espera:
    - $<$ -1: espera a que termine cualquier proceso hijo que pertenezca al grupo de procesos indicado por el valor aboluto de `pid`
    - -1: espera a la finalización de cualquier proceso hijo
    - 0: espera a la finalización de cualquier proceso hijo que pertenezca al mismo grupo de procesos que el proceso padre
    - $>$ 0: espera por la terminación del proceso hijo con este valor de `pid`
  - `status` es un parámetro de salida que codifica el motivo de la terminación del hijo.
    - Macros: WIFEXITED, WIFSIGNALED, ...
  - `options` es una máscara de bits que permite determinar las situaciones que terminan la espera:
    - `WNOHANG`: no ha terminado ningún proceso hijo
    - `WUNTRACED`: un hijo ha sido parado
    - `WCONTINUED`: un hijo se ha reanudado

Si un proceso hijo termina, queda en un estado **zombie** hasta poder entregar el valor de retorno al proceso padre (a través **wait**)

Ejemplo: fork, exec y wait

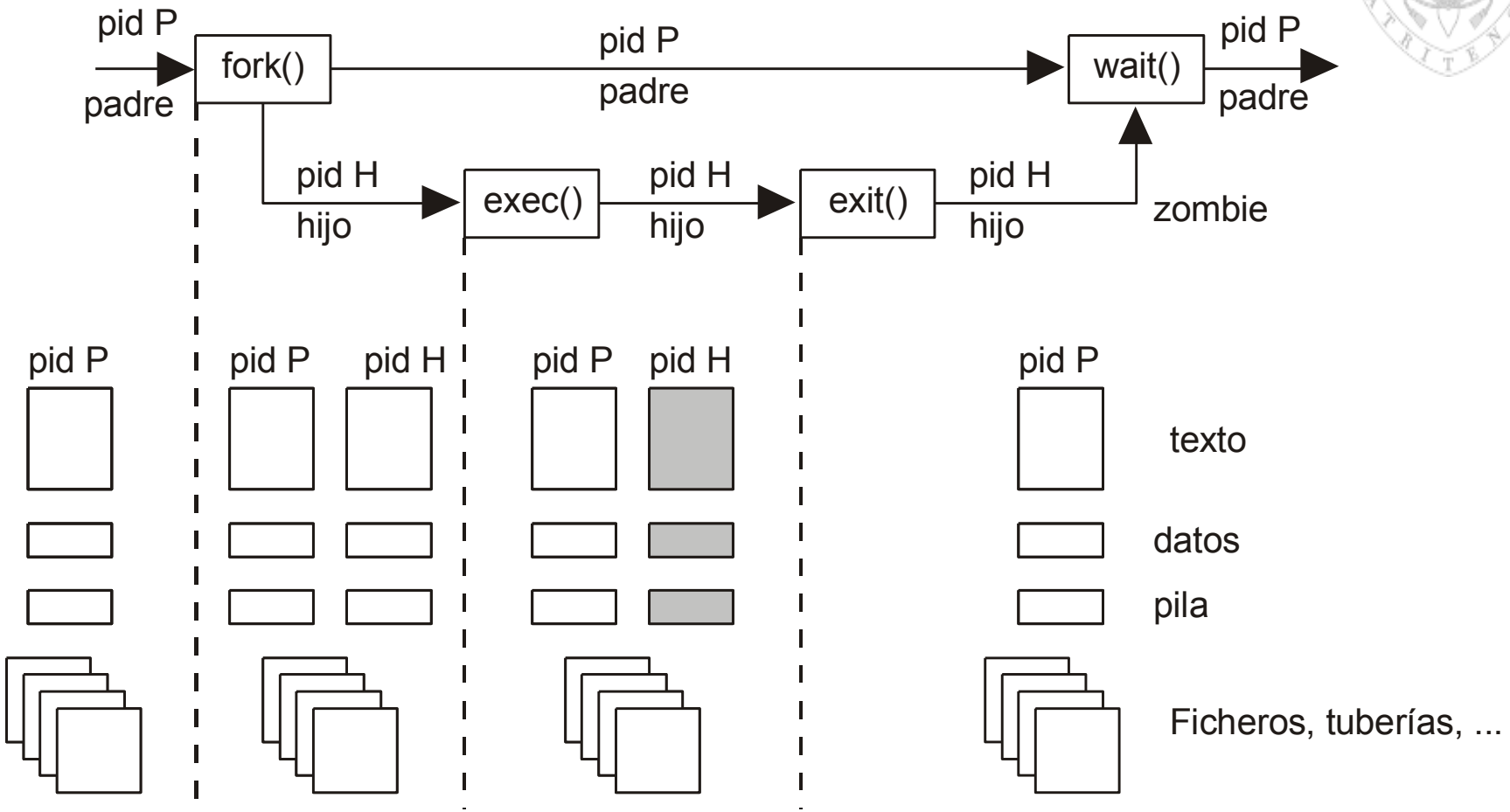
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
/* Ejecutación del programa ls con el flag -l */
int main(void) {
  pid_t pid;
  int status;
  pid = fork();
  if (pid == 0) { /* proceso hijo */
    execlp("ls","ls","-l",NULL);
    perror("");
    return -1;
  } else {          /* proceso padre */
    while (pid != wait(&status));
  }
  return 0;
}
```

Terminación del proceso - exit

Para terminar la ejecución de un proceso debe invocarse la llamada
`int exit(int status)`

- `status`: código de retorno al proceso padre
  - 0 si es una finalización sin error
  - otro si hay error
- El SO liberará todos los recursos no compartidos asociados al proceso
- En un programa C un retorno de `main` supone una llamada a `exit` con el valor devuelto desde el `main`
- Cuidado con los procesos multi-hilo, todos los hilos del proceso terminarán.

Uso normal de procesos



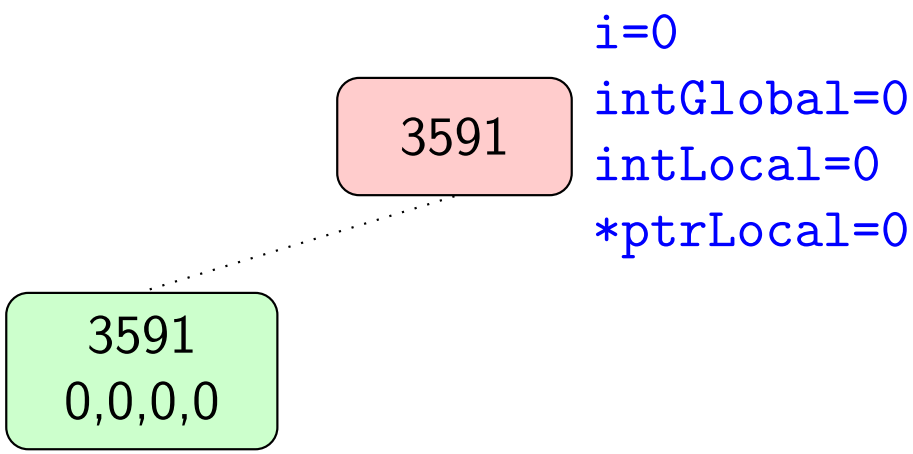
Ejemplo: ejemplo-fork.c 1/3

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

#define N 3
int intGlobal = 0;

int main() {
  pid_t pid;
  int intLocal = 0, i;
  int* ptrLocal = (int*)malloc(sizeof(int));
  *ptrLocal = 0;
```

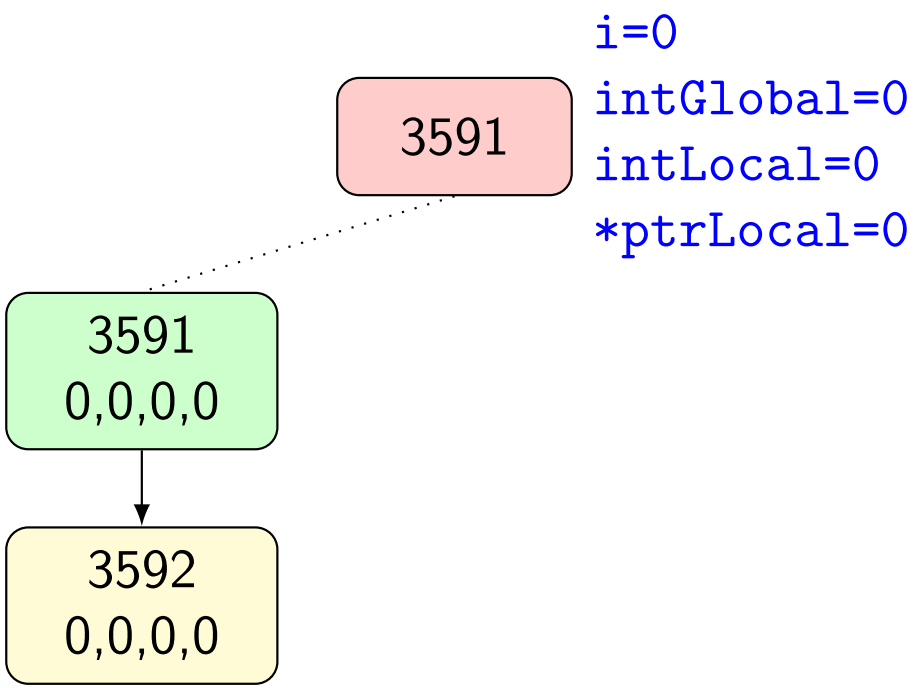
Ejemplo: ejemplo-fork.c 2/3

```
  for (i=0;i<N;i++) {
    pid =fork();
    if (pid==0) { // Este es el proceso hijo
      intGlobal+=2; intLocal+=2; (*ptrLocal) +=2;
    }
    else { // Este es el proceso padre
      intGlobal++; intLocal++; (*ptrLocal)++;
    }
  }

  while (wait(NULL) != -1) ;
  if (errno != ECHILD) {
    printf("ERROR when waiting for childs to finish\n");
    exit(-1);
  }
  exit(0);
}
```

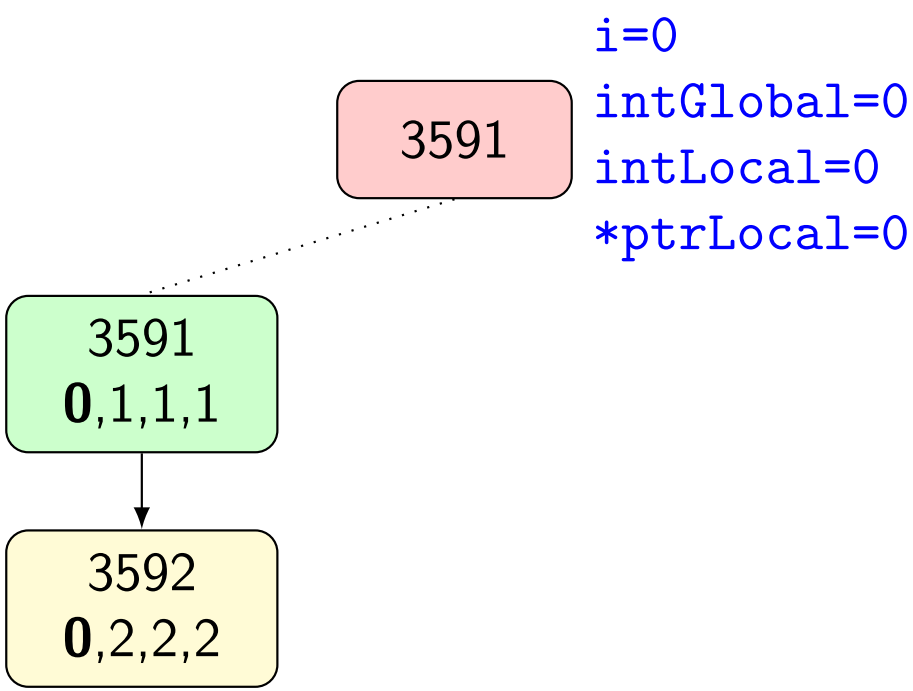
Ejemplo: ejemplo-fork.c 3/3



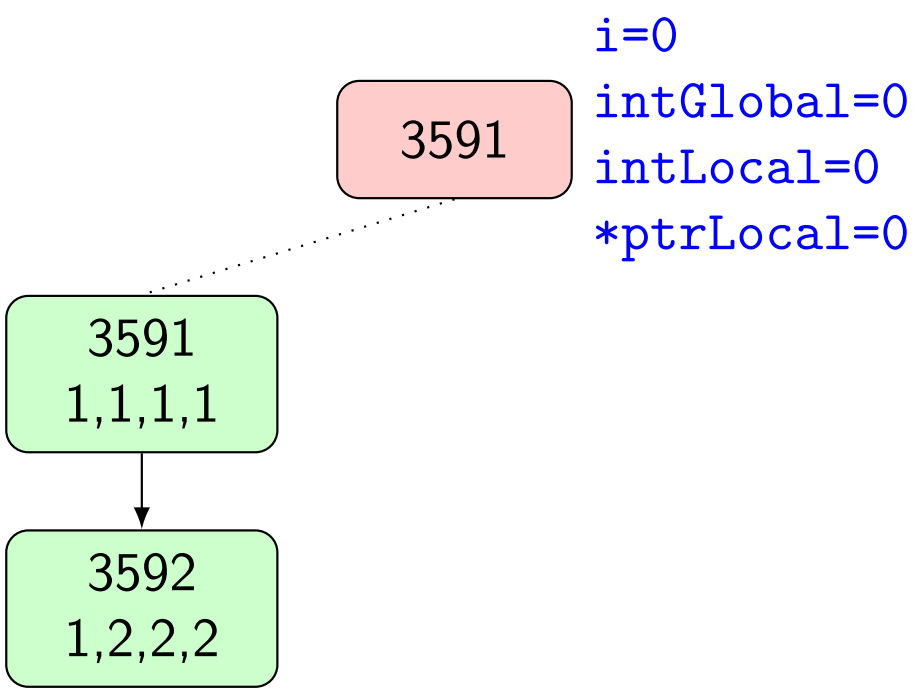
Ejemplo: ejemplo-fork.c 3/3



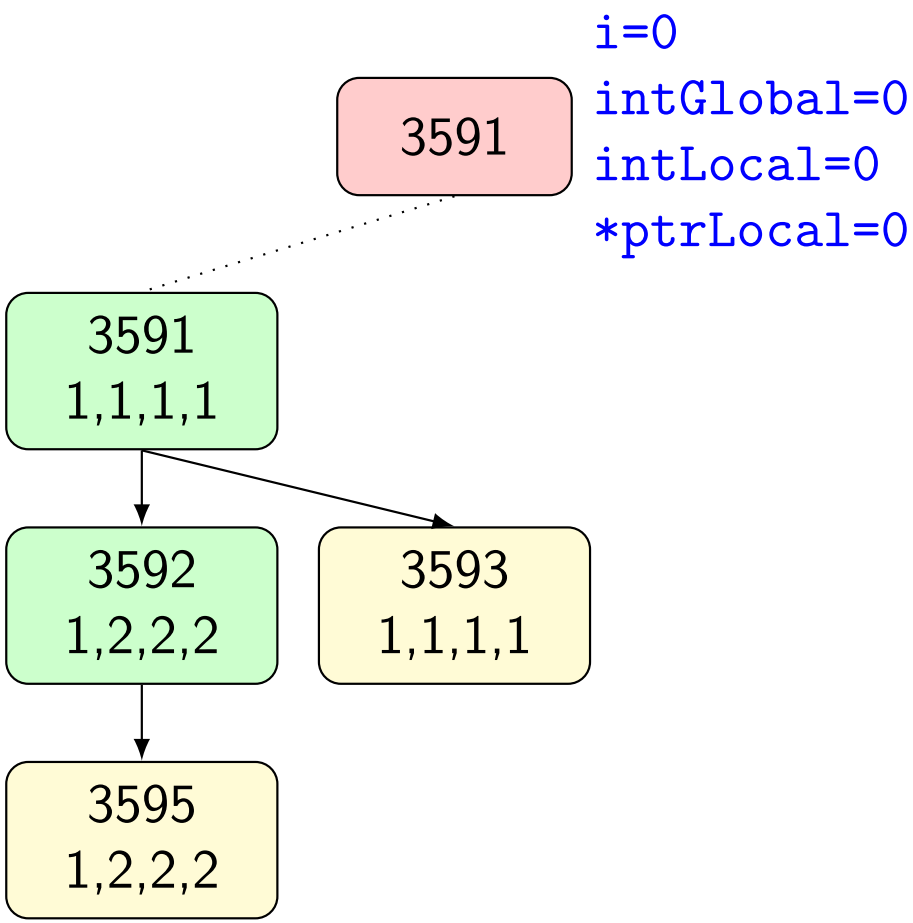
Ejemplo: ejemplo-fork.c 3/3



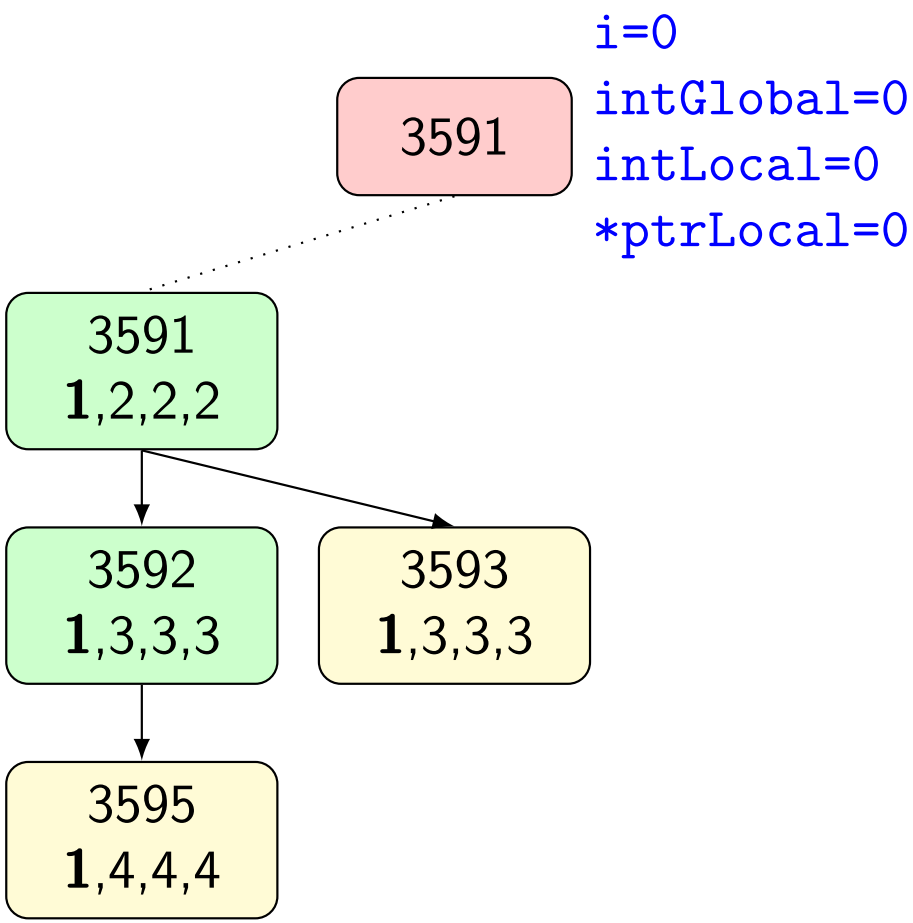
Ejemplo: ejemplo-fork.c 3/3



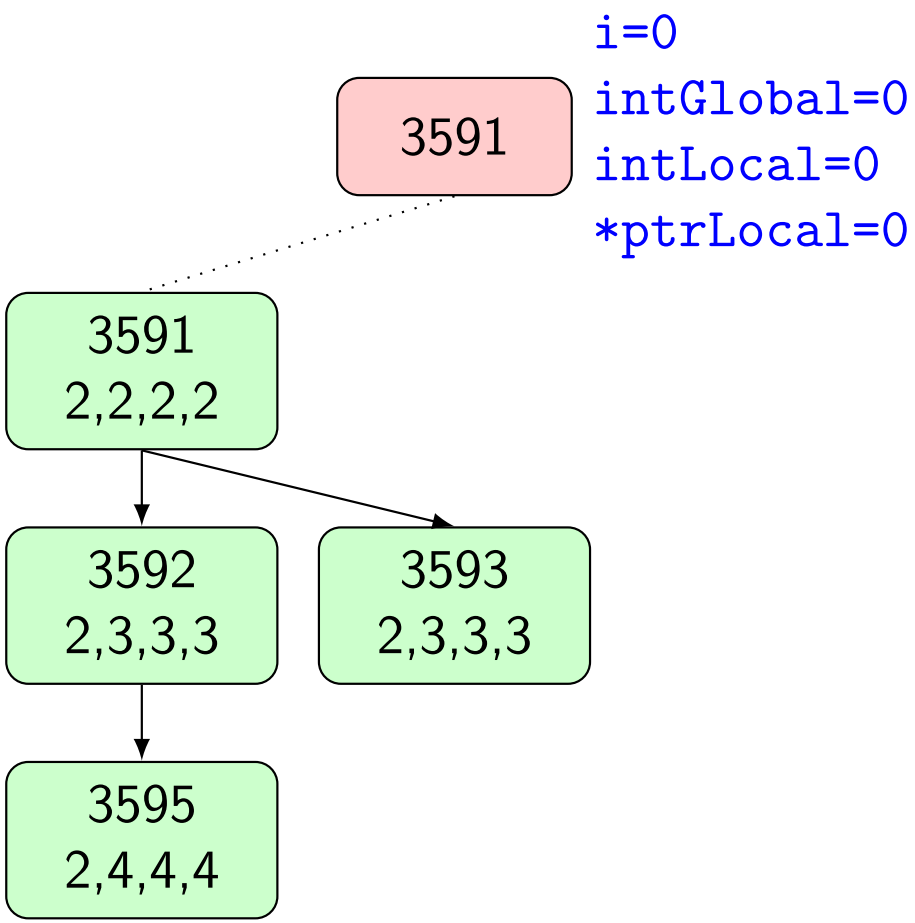
Ejemplo: ejemplo-fork.c 3/3



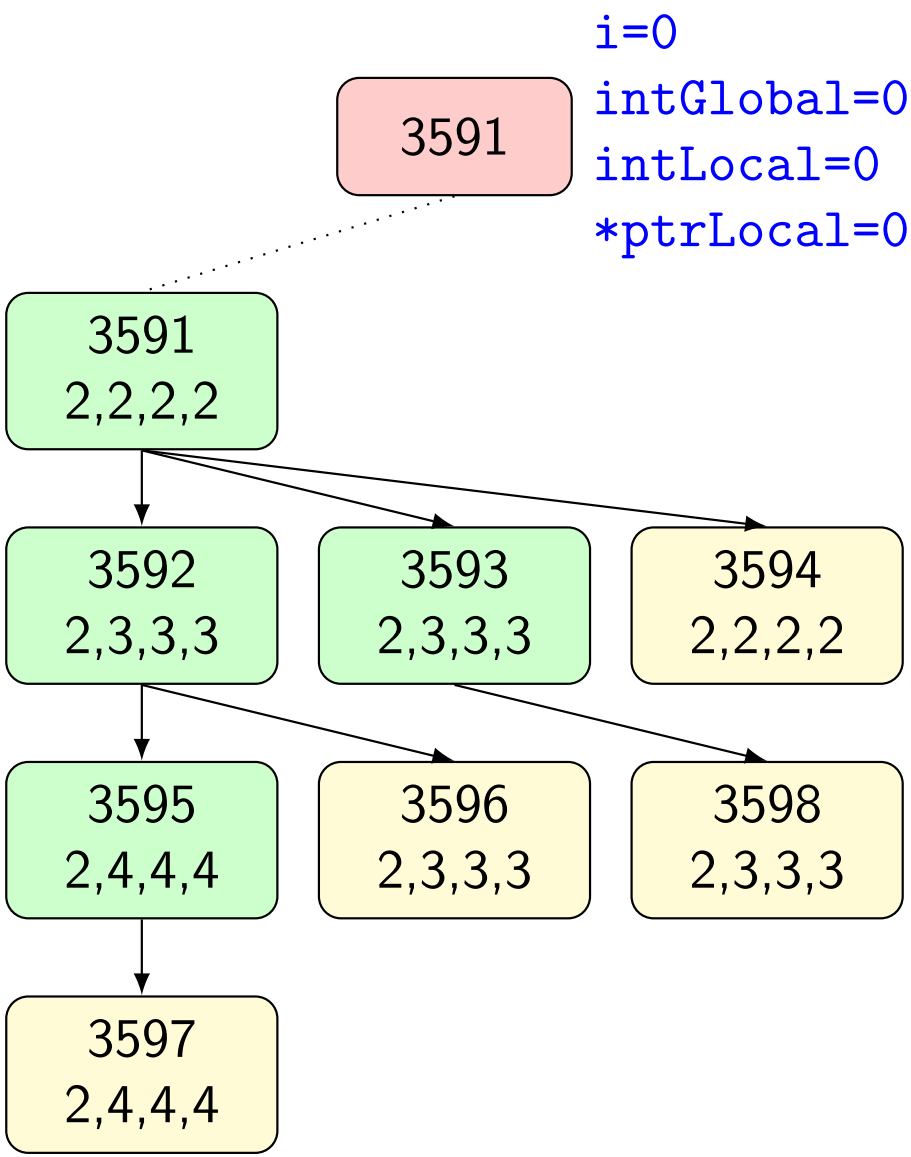
Ejemplo: ejemplo-fork.c 3/3



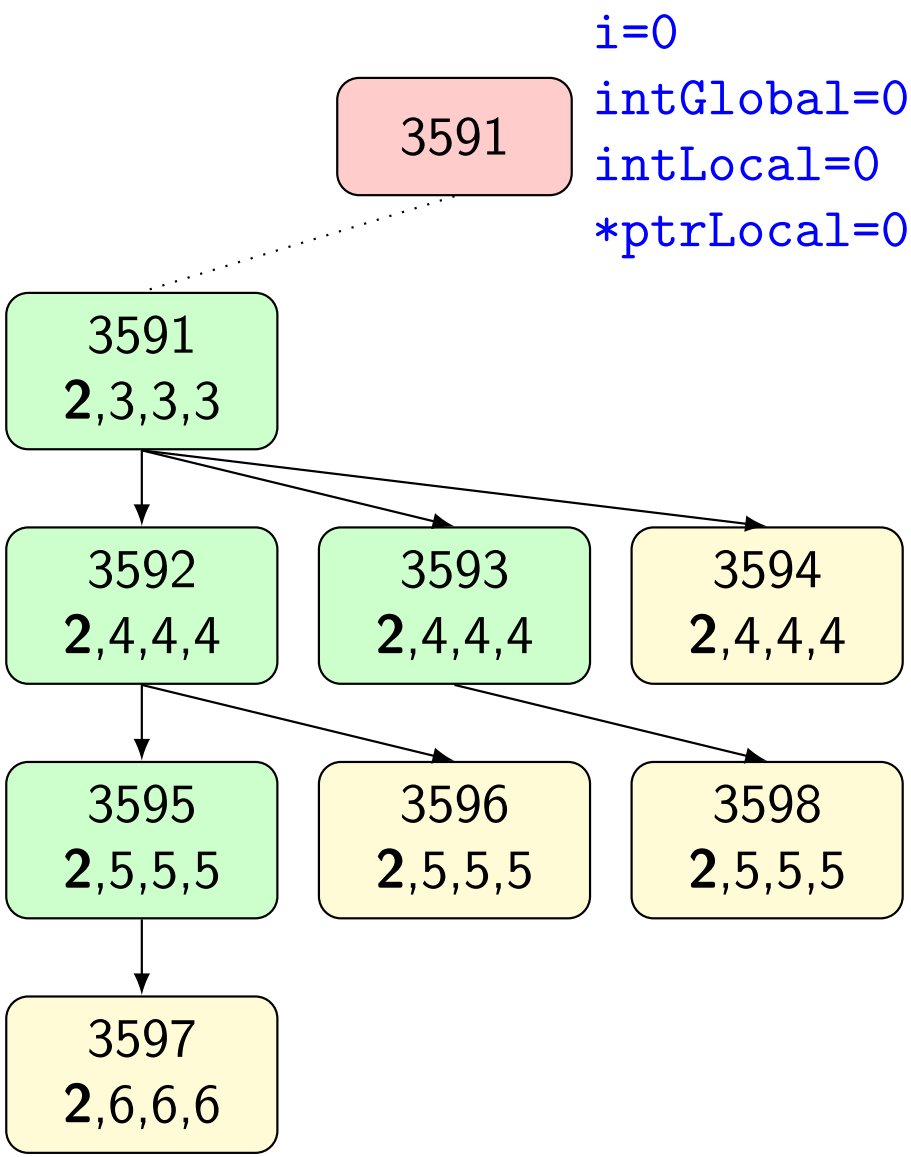
Ejemplo: ejemplo-fork.c 3/3



Ejemplo: ejemplo-fork.c 3/3



Ejemplo: ejemplo-fork.c 3/3



Fin

Envío de señales a un proceso

La llamada al sistema `kill` permite enviar señales a procesos:

```
int kill(pid_t pid, int sig);
```

En función de `pid`:

- `pid > 0`: la señal `sig` es enviada al proceso con este pid
- `pid < -1`: la señal `sig` es enviada a todos los procesos del grupo de procesos `|pid|`
- `pid = 0`: la señal `sig` es enviada a todos los procesos del grupo de procesos del proceso que la envía
- `pid = -1`: la señal `sig` es enviada a todos los procesos a los que el proceso que la invoca tenga permisos para enviar señales, excepto `init` y al propio proceso.

Otras funciones relacionadas son:

- `int raise(int sig)`: para enviarse una señal a sí mismo
- `int killpg(pid_t pgrp, int sig)`: equivalente a `kill(-pgrp, sig)`, envía la señal `sig` a todos los procesos pertenecientes al grupo de procesos `pgrp`.

Agenda

Credenciales de un proceso

El sistema asigna a un proceso las siguientes credenciales:

- Identificadores de proceso (pid) y proceso padre (ppid)
  - Asignados nuevos en fork
- Identificadores de grupo de procesos (pgid) y sesión (sid)
  - Heredados del padre en fork, preservados en exec
  - Identificadores creados para el control de trabajos del shell
- Identificadores de usuario (uid) y grupo (gid)
  - Determinan el usuario y grupo propietarios del proceso
  - Heredados del padre en fork
- Identificadores de usuario efectivo (euid) y grupo efectivo (egid)
  - Usuario y grupo con el que se comprueban los permisos de acceso a recursos compartidos
  - Inicialmente son el mismo que el uid y el gid respectivamente
- Identificadores set-user-id y set-group-id
  - Adquiridos en exec al ejecutar ficheros con los bits setuid y setgid activos
  - Un proceso puede cambiar su euid entre el uid real y el set-uid
  - y su egid entre el gid real y el set-gid

Jerarquía de procesos Linux (UNIX)

Árbol de procesos

```
# pstree -p

systemd(1)-+-ModemManager(3159)-+-{ModemManager}(3255)
           |                    `-{ModemManager}(3258)
           |-NetworkManager(3188)-+-dhclient(5944)
           |                      |-{NetworkManager}(3261)
           |                      `-{NetworkManager}(3265)
           |-accounts-daemon(3168)-+-{accounts-daemon}(3190)

...
```

- El primer proceso creado en el sistema es **systemd** (`init`), pid = 1:
  - Encargado de arrancar el SO y lanzar los demás procesos
  - Todos los procesos del sistema son sus sucesores
  - Al finalizar su tarea de levantar el sistema se queda esperando a que terminen sus hijos
  - Si un proceso padre termina antes que sus procesos hijo, los procesos hijos son *heredados* por proceso antecesor más cercano marcado como `subreaper` o por **systemd**.
  - En Linux podemos ver el árbol de procesos con **pstree**

Sesiones

Los procesos están organizados por sesiones

- El ID de la sesión es el pid del proceso que la creó utilizando la llamada al sistema `setsid()`
  - Es el líder de la sesión
  - Normalmente es el shell
- Cada sesión tiene asociado a un terminal de control
  - Es la entrada y salida estándar para los procesos de la sesión
  - Se establece cuando el líder de la sesión abre por primera vez un terminal
  - El shell establece qué grupo de procesos es el que controla el terminal (trabajo en foreground) usando la función: `tcsetpgrp`.
    - Sólo el grupo de procesos en foreground puede leer del terminal, los procesos en background reciben la señal SIGTTIN que para el proceso.
    - Si el terminal tiene el flag TOSTOP activo, sólo el grupo de procesos en foreground puede escribir en el terminal, los procesos en background reciben la señal SIGTTOU que para el proceso.
- Los procesos heredan la sesión de su padre
  - Pueden salir de ella creando su propia sesión (`setsid`)
  - En este caso pierde el terminal de control

Grupos de procesos (1/2)

Los grupos de procesos son utilizados por el shell para crear trabajos:

- una sentencia del shell, posiblemente compuesta de varios procesos encadenados con pipes.

Trabajos (JOBS)

```
# touch kk
# tail -f kk | cat - &
[1] 14038
# jobs -l
[1]+ 14037 Ejecutando              tail -f kk
     14038                       | cat - &
# ps -j
   PID   PGID    SID TTY          TIME CMD
 10822  10822  10822 pts/0    00:00:00 bash
 14037  14037  10822 pts/0    00:00:00 tail
 14038  14037  10822 pts/0    00:00:00 cat
 14064  14064  10822 pts/0    00:00:00 ps
```

Grupos de procesos (2/2)

Los procesos pueden asignarse a un grupo mediante la llamada al sistema:

```
int setpgid(pid_t pid, pid_t pgid)
```

Con las siguientes reglas:

- un proceso sólo puede cambiar su id de grupo o el de algún hijo
- un proceso líder de sesión no puede cambiar su id de grupo
- si se cambia el id de grupo de un proceso, el líder del nuevo grupo tiene que estar en la misma sesión que el proceso
- Un proceso que invoque `setpgid(0,0)` se convierte en líder de su propio grupo (con id el pid del proceso)

Podemos enviar señales a todos los procesos del grupo/trabajo:

- Con la utilidad `kill`: `kill SIGNAL %jobid`
- Con la llamada al sistema kill y un número de proceso negativo

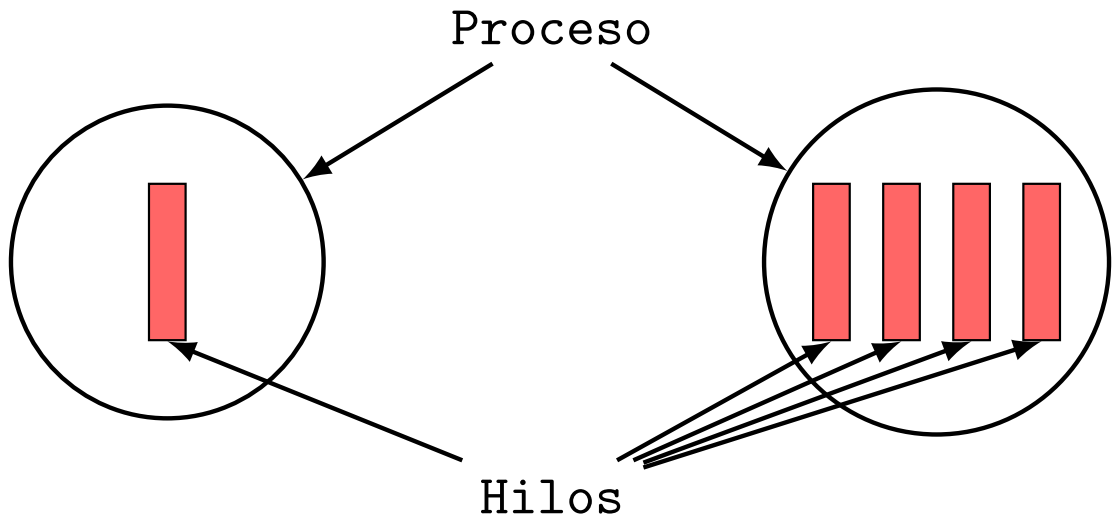
Utilidades: procesos y trabajos

Se recomienda consultar las páginas de manual de:

- **ps**: lista los procesos activos del sistema
- **pgrep**: equivalente a **ps** | **grep**
- **top**: lista ordenada e interactiva de los procesos del sistema
- **kill**: envio de una señal a un proceso
  - o trabajo con %N donde N es el número de trabajo
- **killall**: envio de una señal a un proceso por nombre
- **pidof**: obtiene el pid de un proceso por nombre (filtra la salida ps)
- **pkill**: permite mandar una señal a un proceso por el nombre del commando ejecutado (usa grep y pidof)
- **pstree**: muestra el árbol de procesos
- **jobs**: lista trabajos activos en el shell
- **wait**: permite esperar la finalización de un proceso o un trabajo
- **nohup**: permite lanzar un proceso que ignore la señal SIGHUP

Agenda

Concepto de Hilo (Thread)



Cada hilo tiene su propio:

- Un contexto de ejecución
- Una región de pila, en el espacio de direcciones virtuales del proceso
- Afinidad, política de planificación y prioridad de tiempo real
- *variable errno*

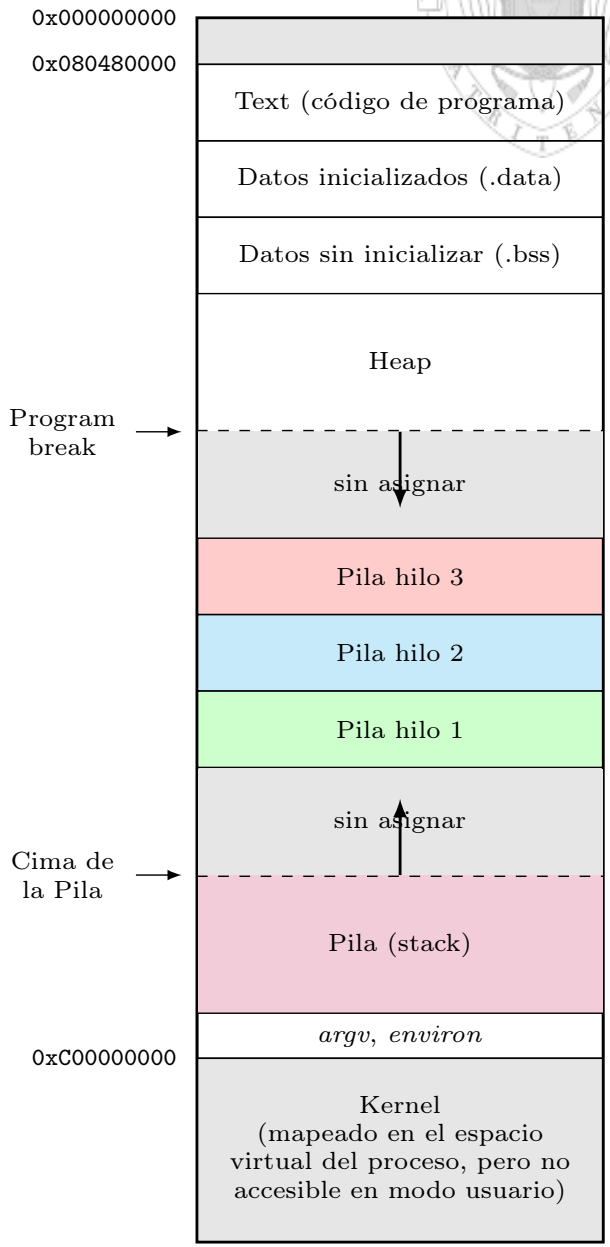
Comparten los recursos del proceso:

- Espacio de direcciones
- Tabla de ficheros abiertos
- Credenciales del proceso
- Terminal de control
- Timers, semáforos y manejadores de señal
- Límites de recursos
- Valor de nice (no en Linux)

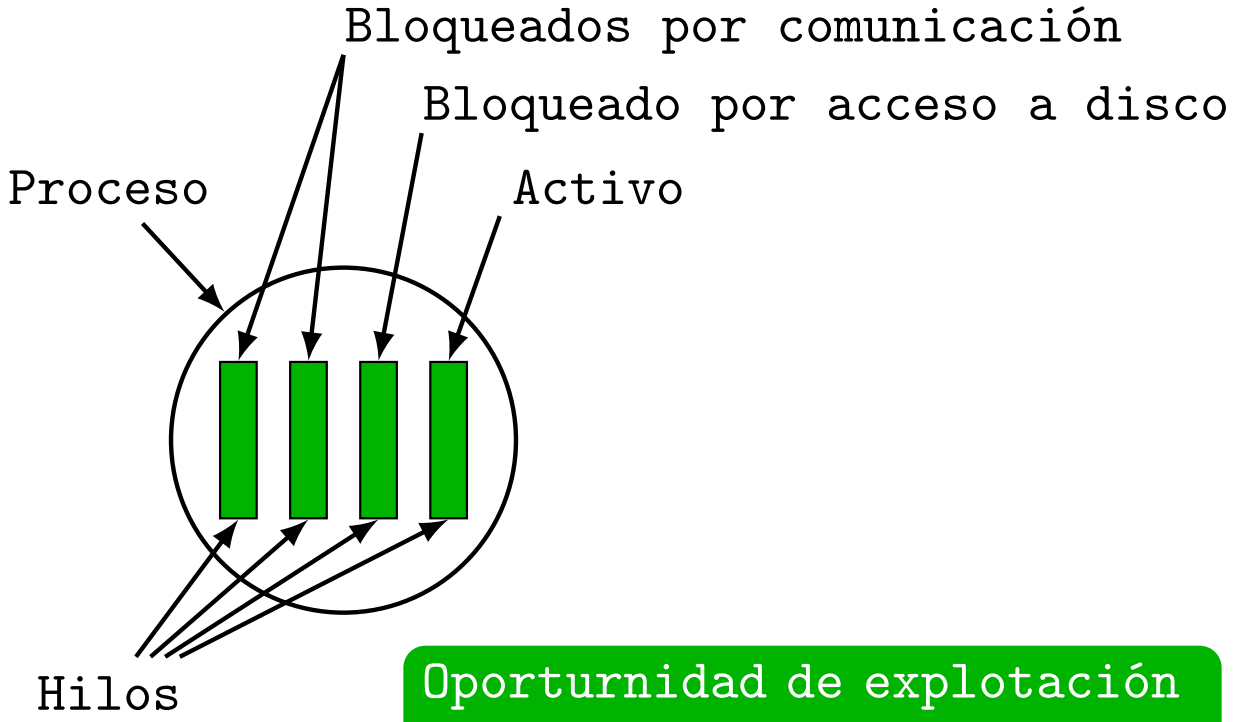
Nuevos recursos por cada hilo

El sistema crea algunos recursos privados para los hilos:

- Una región de memoria para ubicar la pila de llamadas de cada hilo creado en modo usuario
  - El hilo principal, creado con el proceso, usa la región de pila original del proceso
- Otra región por hilo para ubicar las pilas en modo kernel
- Un bloque de control del hilo (BCT), accesible desde el BCP
  - Contiene el contexto de ejecución (reg. arquitectónicos),
  - la política y el estado de planificación,
  - la prioridad de tiempo real,
  - y la afinidad del hilo.



Un estado de planificación por hilo



Oportunidad de explotación de paralelismo dentro del proceso

Hilos vs Procesos

Ventajas del uso de hilos:

- Menor coste de creación, destrucción y planificación
- Menor coste del cambio de contexto entre hilos del mismo proceso
  - No es necesario cambiar el espacio de direcciones activo
- Los hilos de un mismo proceso comparten fácilmente memoria
  - Variables globales del proceso
  - Heap del proceso
  - Otras regiones del proceso

Cosas a tener en cuenta:

- Funciones reentrantes:
  - Puede ser interrumpida en mitad de la ejecución (incluso por `isr`) y ser invocada de nuevo antes de que finalice la invocación anterior.
- Funciones `thread-safe`:
  - Sólo manipulan recursos compartidos de manera controlada, sincronizando el acceso entre múltiples hilos.
- Sincronización en acceso a variables globales

Servicios POSIX para hilos

- Creación de un hilo con función de entrada `func`

```
int pthread_create(pthread_t *thread, const pthread_attr_t  *attr,
                   void *(*func)(void *), void *arg)
```

  - Atributos: tamaño de la pila, prioridad, política de planificación,...
  - Existen llamadas para modificar los atributos

- Terminación del hilo y retorno de valor de terminación

```
int pthread_exit(void *value)
```

- Esperar a la terminación de un hilo y obtener el valor de terminación

```
int pthread_join(pthread_t thread, void **value)
```

- Obtención del identificador de hilo

```
pthread_t pthread_self(void)
```

Ejemplo pthreads 1/2

```c
#include <stdio.h>
#include <pthread.h>
#define NTH 10

struct tharg {
    int i;
    char *msg;
};

void* foo(void* arg)  {
    struct tharg *tharg = (struct tharg *) arg;
    printf("Thread %d with pthread id %ld says %s\n", tharg->i,
      (long)pthread_self(), tharg->msg);
    pthread_exit(0);
}
```

Ejemplo pthreads 2/2

```
int main(void){
    int j;
    pthread_attr_t attr;
    struct tharg args[NTH];
    pthread_t thid[NTH];

    pthread_attr_init(&attr);
    for(j = 0; j < NTH; j ++) {
        args[j].i = j;
        args[j].msg = "Hello!";
        pthread_create(&thid[j], &attr,  foo, &args[j]);
    }

    for(j = 0; j < NTH; j ++)
        pthread_join(thid[j], NULL);
    return 0;
}
```