Sistemas Operativos

Gesitón de Procesos

Conceptos de Proceso e Hilo

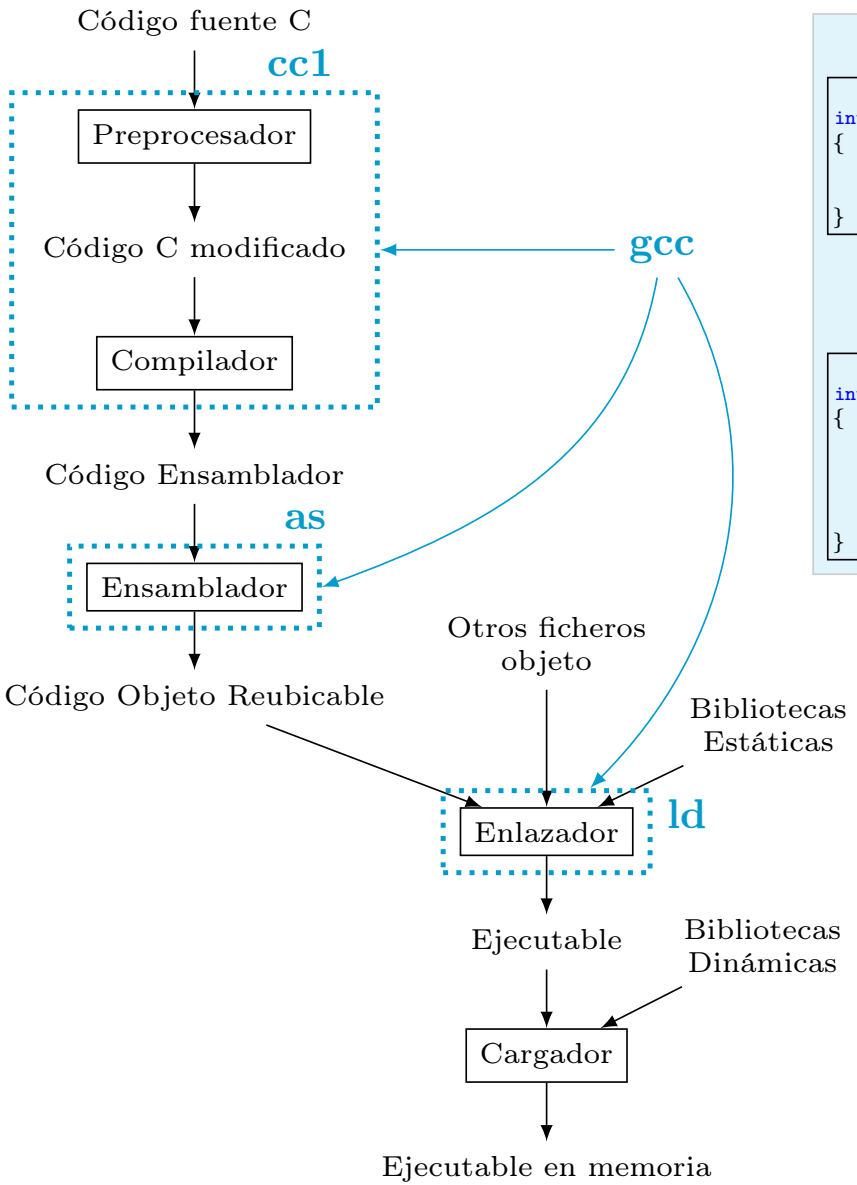
Agenda

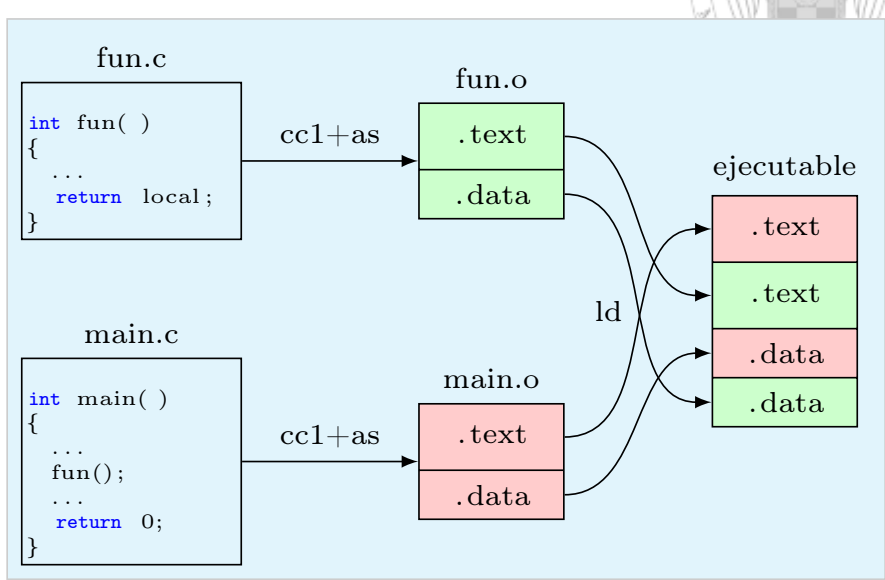
Agenda

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

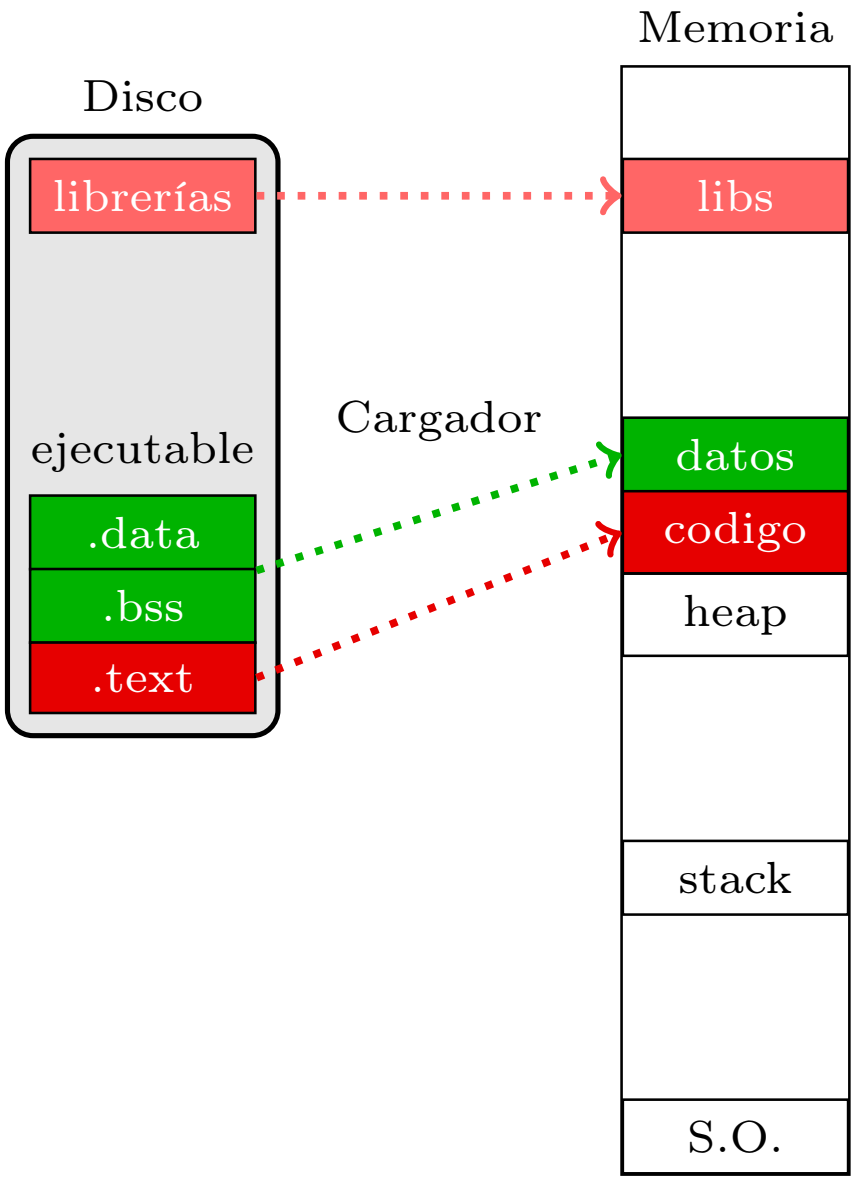

Memoria
Disco
librerías
libs
Cargador
ejecutable
datos
.data
codigo
.bss
heap
.text
stack
S.O.


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

- Formada por el conjunto de regiones de memoria que un proceso está autorizado a utilizar
  - Región de código
  - Región de datos
  - Pila
  - Heap
  - ...
- Si un proceso genera una dirección que esta fuera del espacio de direcciones el HW (MMU) genera una excepción que el SO captura

/proc/pid/maps

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. Consultar `man proc`.

Tabla de páginas

Permite la traducción de direcciones virtuales a direcciones físicas

. . . Módulo de Gestión de Memoria

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)

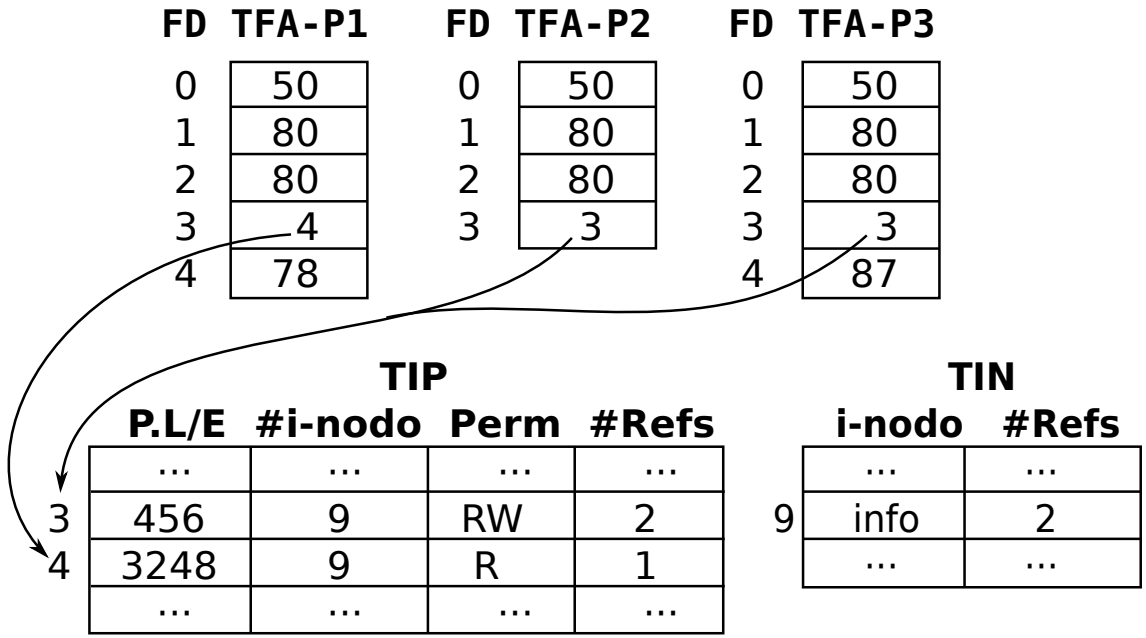
Información de planificación del proceso

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)

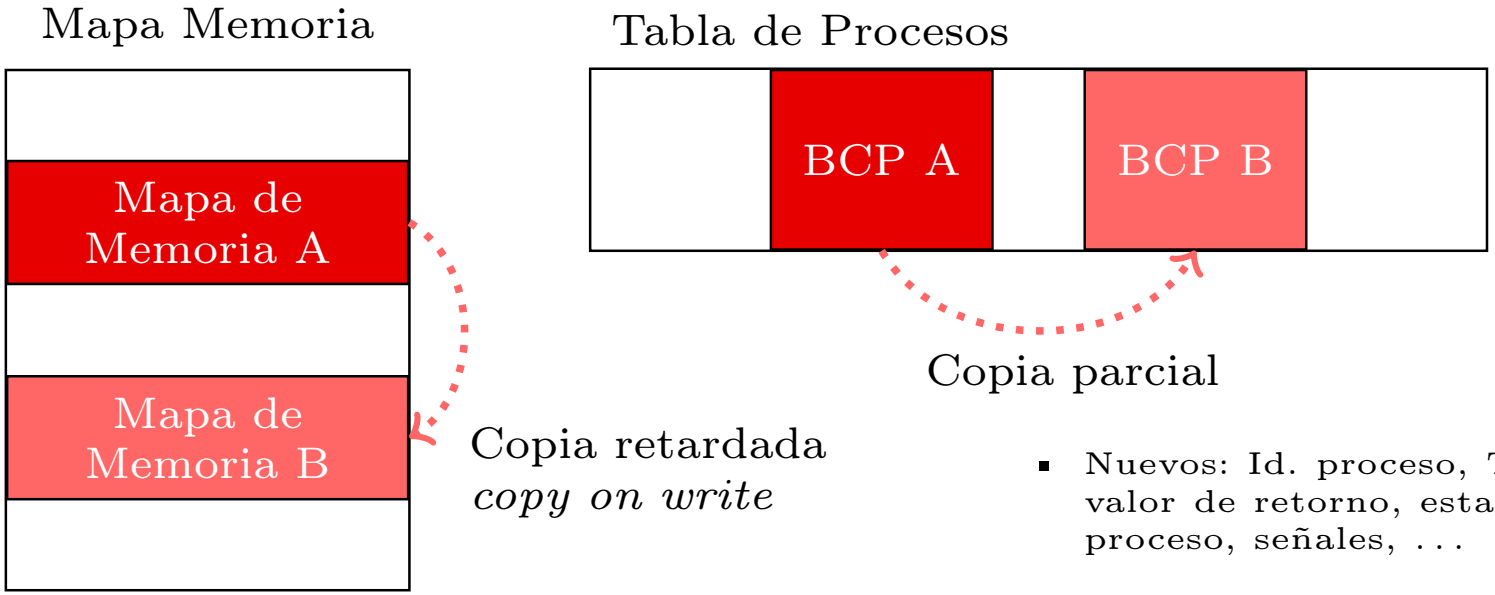
- En POSIX los procesos pueden recibir señales
  - de otros procesos: `int kill(pid_t pid, int sig)`
  - del propio SO
- El SO guarda una máscara de señales pendientes por atender para cada proceso (no encola varias señales del mismo tipo)
- Un proceso puede:
  - bloquearse a la espera de la recepción 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.



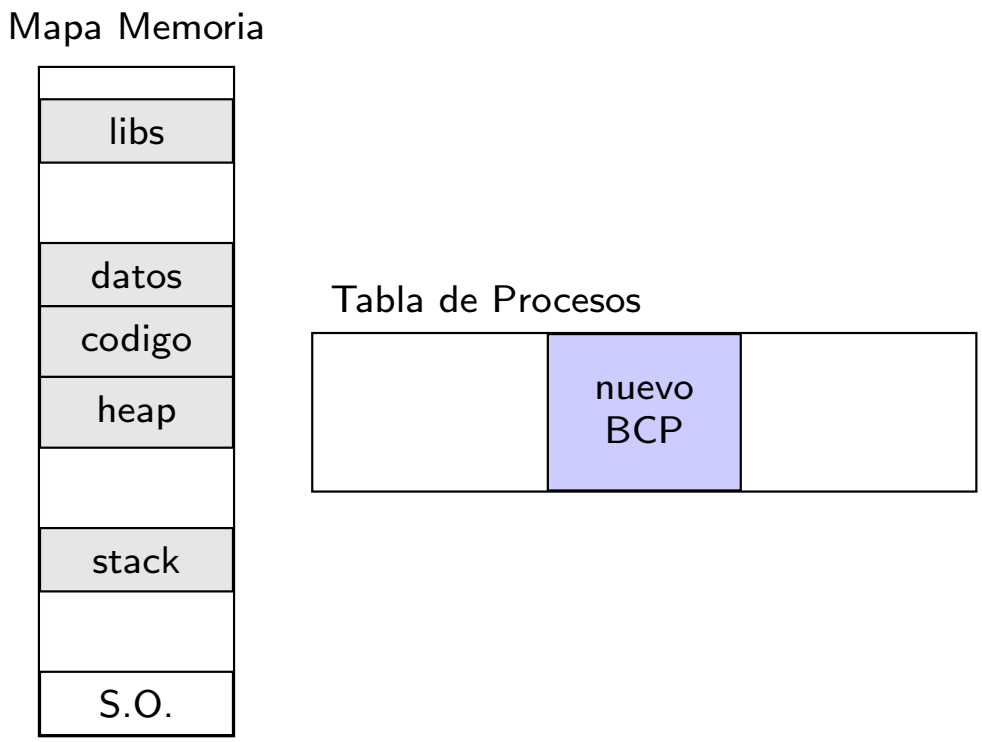
- Nuevos: Id. proceso, TP, valor de retorno, estado del proceso, señales, ...
- Copiados: TFA, Id. usuario, resto del estado arquitectónico, prioridad, ...

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 */
          ...
     }
     ...
}
```

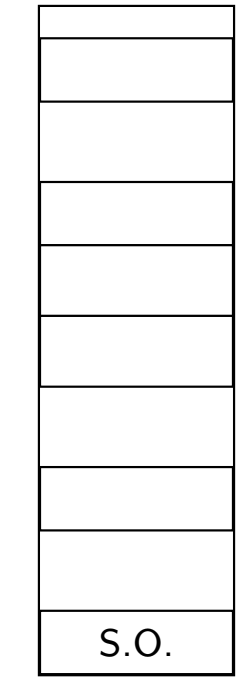
Cambio de programa - exec

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

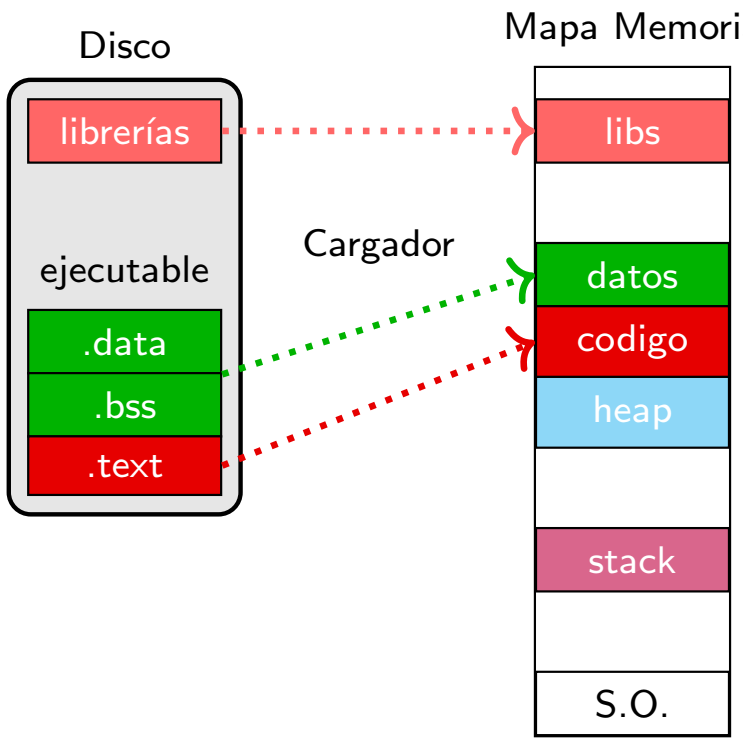
La familia de llamadas al sistema `int exec*(const char *path, ...)` permite cambiar el mapa de memoria del proceso por el de un nuevo programa con argumentos de llamada



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

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)

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 - wait

Un proceso puede bloquearse a la espera de que termine alguno de sus procesos hijo, utilizando alguna de las siguientes llamadas al sistema:

- `pid_t wait(int *status)`
  espera la finalización de cualquiera de sus procesos hijo
- `pid_t waitpid(pid_t pid, int *status, int options)`
  espera por la terminación de un proceso hijo determinado identificado por su `pid`

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

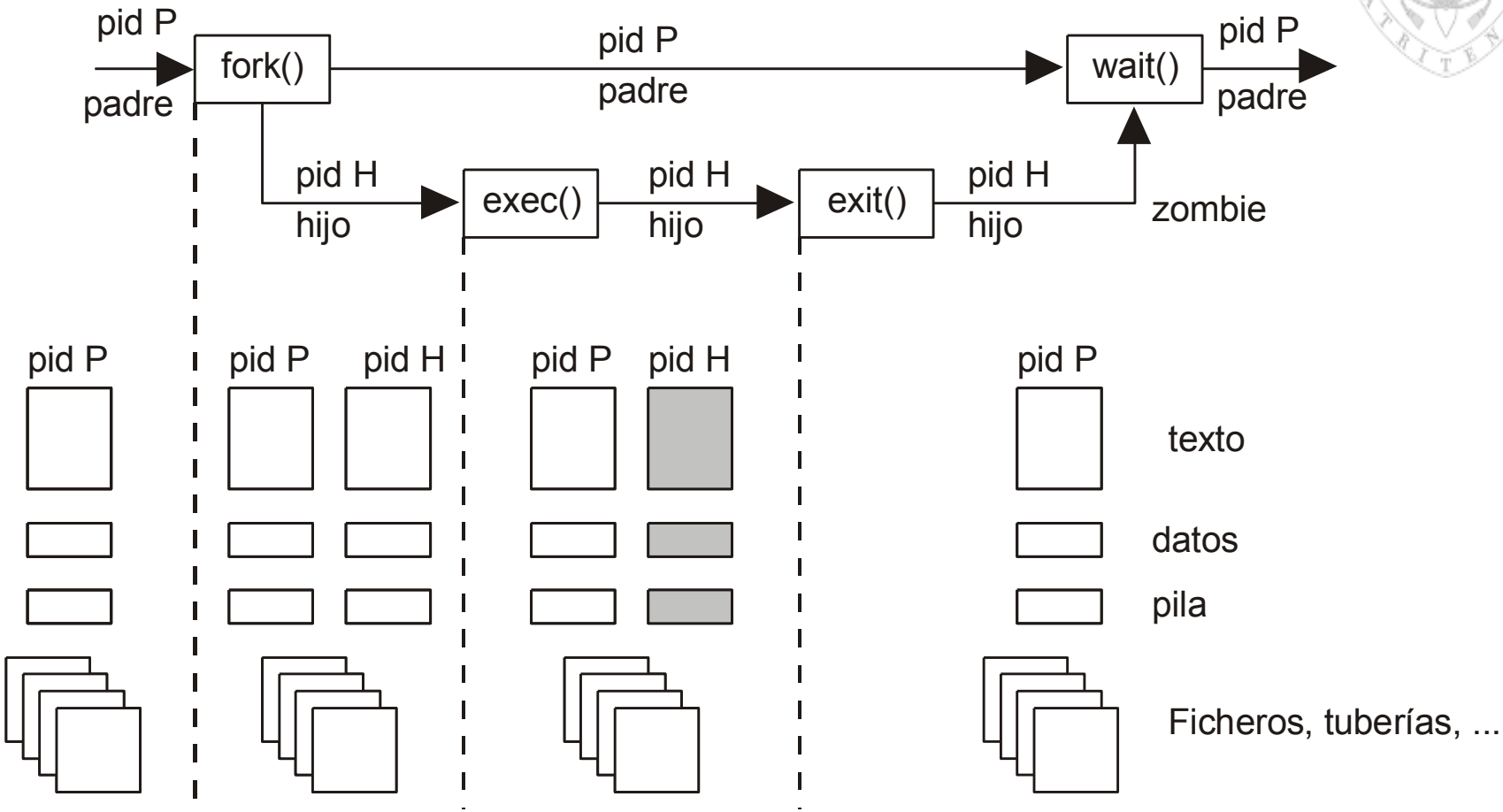
```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
/* Ejecutación del programa ls con el flag -l */
int main(void) {
    pid_t pid;
    int status;
    pid = fork();
    if (pid == 0) { /* proceso hijo */
      execlp("ls","ls","-l",NULL);
      perror("");
    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



Agenda

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 creo utilizando la llamada al sistema `setsid()`
  - Es el líder de la sesión
- Todos los procesos creados a continuación pertenecen a la sesión, a menos que se quiten de la sesión creando su propia sesión (`setsid`)

Sesiones

- Los procesos están organizados por sesiones
- El ID de la sesión es el pid del proceso que la creo utilizando la llamada al sistema `setsid()`
  - Es el líder de la sesión
- Todos los procesos creados a continuación pertenecen a la sesión, a menos que se quiten de la sesión creando su propia sesión (`setsid`)

Terminal de Control

Cada sesión está asociada a un terminal de control, que es la entrada y salida estándar para los procesos de la sesión

- Cuando un proceso se ejecuta en background ya no puede recibir datos del terminal pero si puede escribir datos en él

Grupos de procesos

- Los procesos pueden asignarse a grupos, 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 lider 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 lider de su propio grupo (con id el pid del proceso)

Grupos de procesos y Trabajos

Cuando el shell ejecuta un trabajo pone a todos los procesos del trabajo en el mismo grupo:

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
```

Podemos enviar señales a todos los procesos del grupo/trabajo (`kill SIGNAL %jobid`)

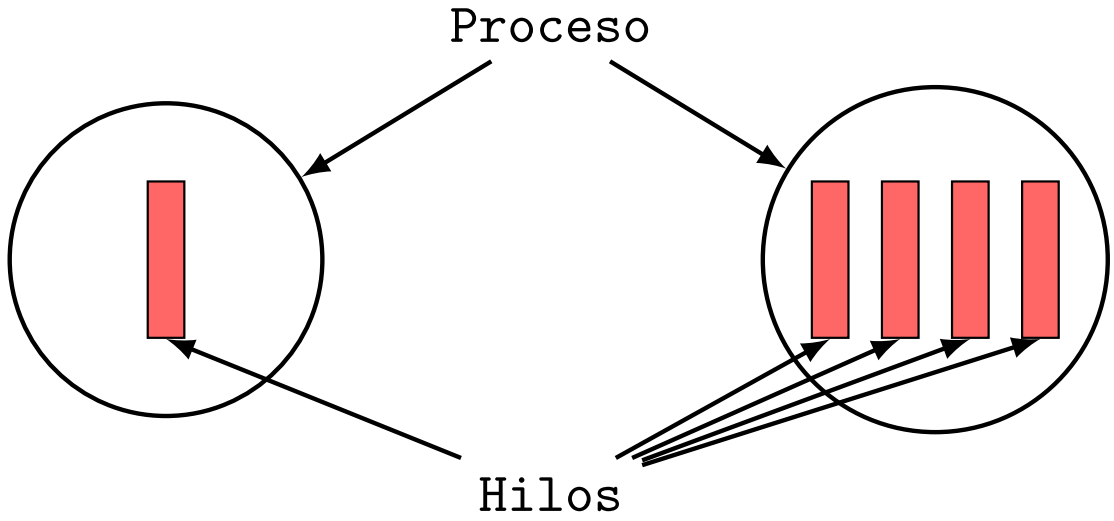
Utilidades: procesos y trabajos

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)



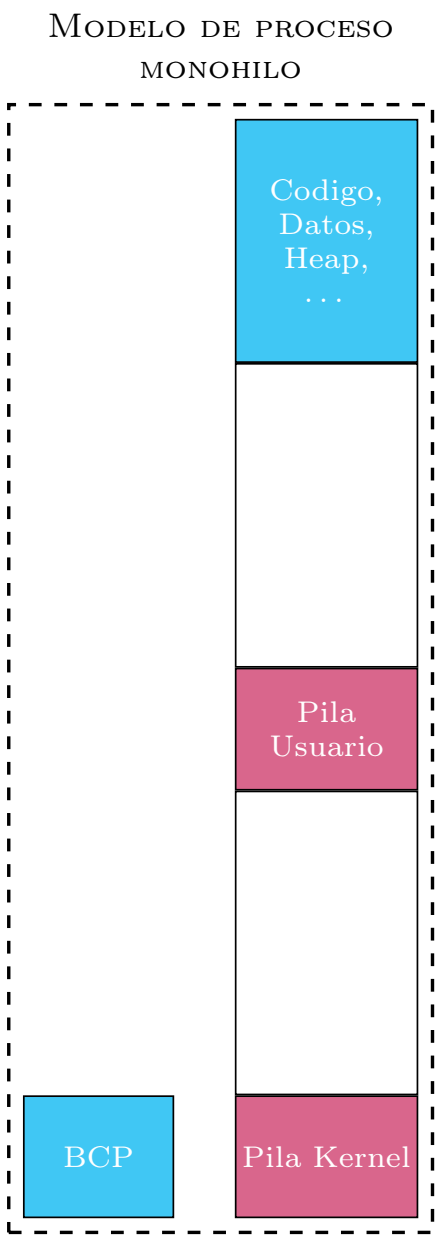
Cada hilo tiene su propio:

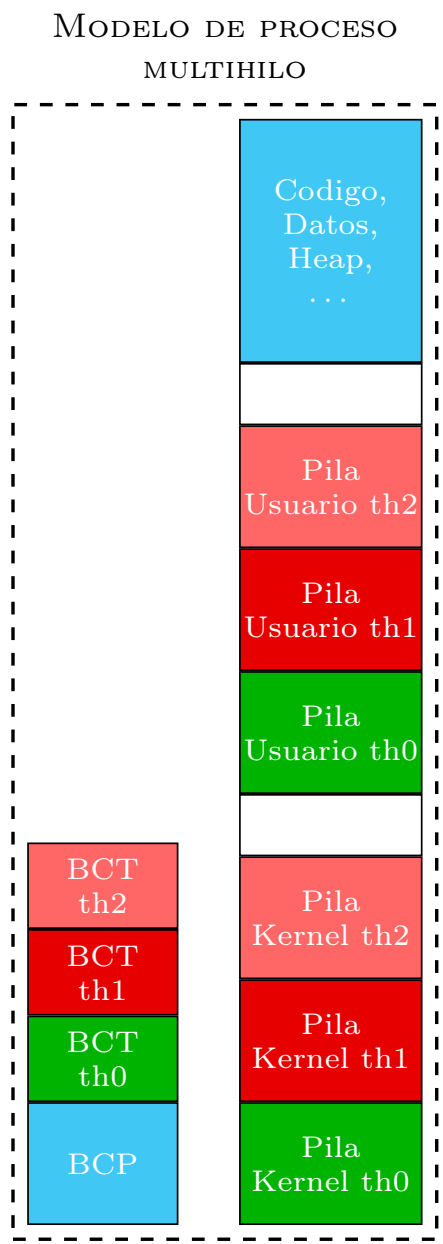
- Estado del procesador (PC, ...)
- Estado de planificación
- Pila

Único por proceso:

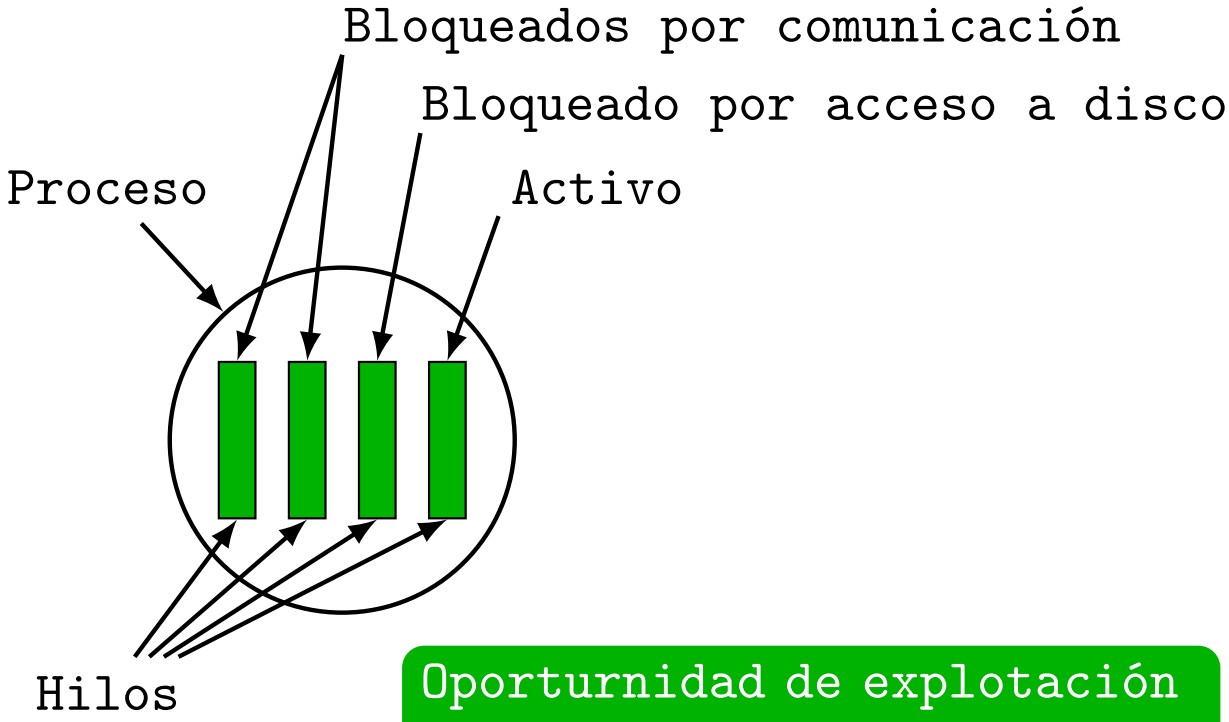
- Imagen de Memoria (TP, variables globales, heap, ...)
- TFA
- Señales, ...

Proceso Multihilo





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

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

void* func(void* arg)  {
  printf("Thread %ld\n",(long unsigned int)pthread_self());
  pthread_exit(0);
}

int main(void){
  int j;
  pthread_attr_t attr;
  pthread_t thid[MAX_THREADS];
  pthread_attr_init(&attr);

  for(j = 0; j < MAX_THREADS; j ++)
    pthread_create(&thid[j], &attr,  func, NULL);

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

Agenda

FAQ

- ¿Cómo es posible que se ejecuten a la vez más procesos/hilos que CPUs hay en la máquina?
- ¿Cómo hace el SO para dejar de ejecutar un proceso/hilo y comenzar a ejecutar otro?
- ¿Cómo pueden comunicarse los procesos/hilos?
- ¿No hay problemas en el acceso simultáneo a recursos? ¿Cómo se resuelven?