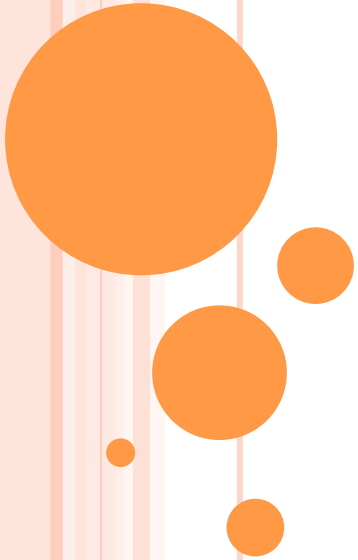
Práctica 5 : procesos

# Fundamentos de los Sistemas Operativos

*Grado en Ingeniería Informática*

Rubén García Rodríguez
Alexis Quesada Arencibia
Eduardo Rodríguez Barrera
Francisco J. Santana Pérez
José Miguel Santos Espino

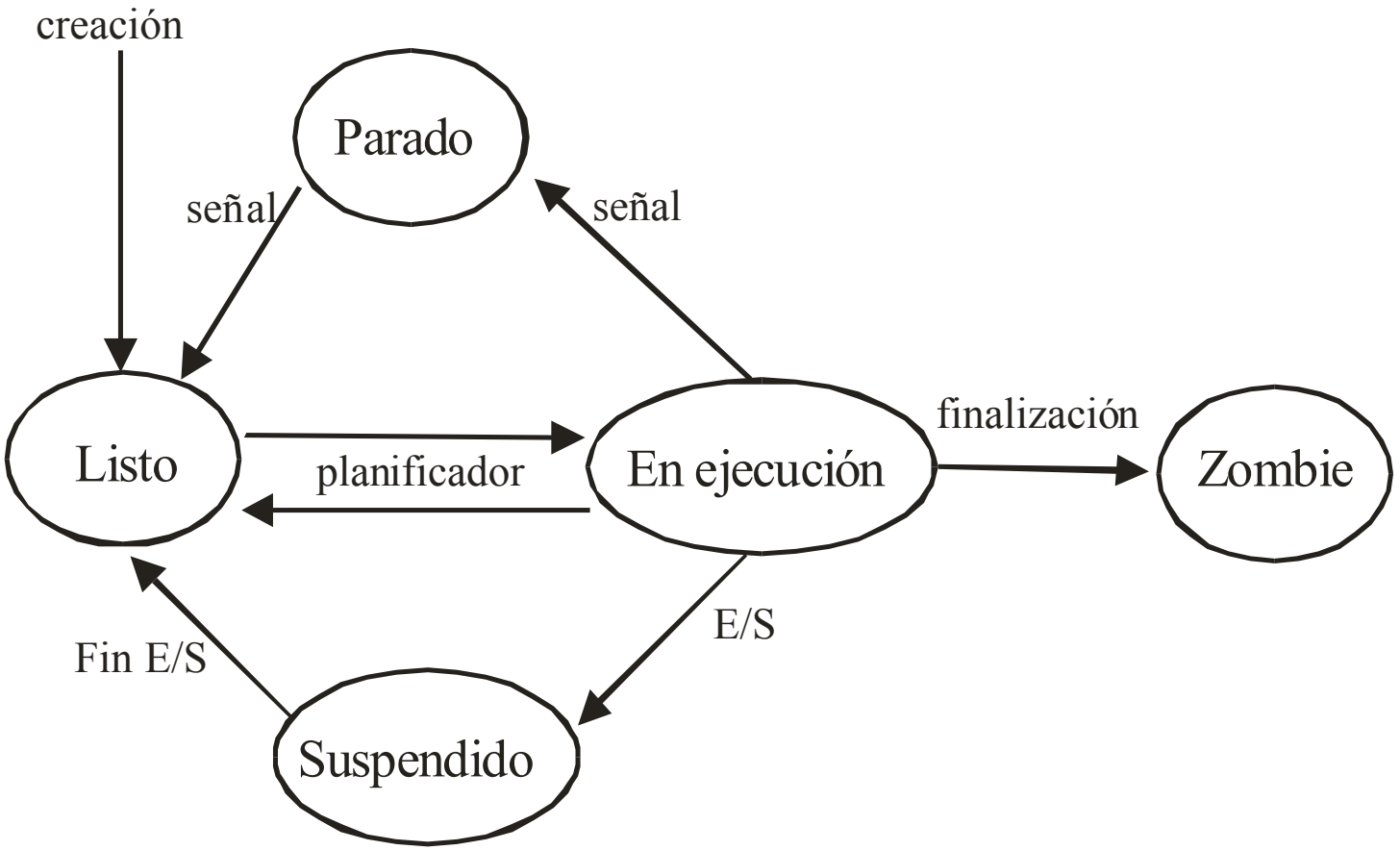


UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela de Ingeniería Informática

# CONCEPTO DE PROCESO

- Un programa es un conjunto de instrucciones almacenadas en disco
- En UNIX, a un programa que se ha cargado en memoria para ejecución se le denomina proceso
- Todo proceso tiene asociado en el sistema un identificador numérico único (PID)

# ESTADOS DE UN PROCESO



# ATRIBUTOS DE UN PROCESO

- Estado
- **PID**
- **PPID** (Id de su padre)
- Valor de los registros
- Identidad del usuario que lo ejecuta
- Prioridad
- Información sobre espacio de direcciones (segmentos de datos, código, pila)
- Información sobre la E/S realizada por el proceso (descriptores de archivo abiertos, dir. actual, etc.)
- Contabilidad de recursos utilizados

¿Para qué usar el PID y el PPID?

-creación de archivos termporales

- identificar qué proceso escribe un registro en un archivo

# CONCEPTO DE PROCESO

- Identificador de usuario (UID): el identificador del usuario que ha lanzado el programa

- Identificador de usuario efectivo (EUID): puede ser distinto del de usuario, p.ej en los programas que poseen el bit *setuid (bit "s").* Se usa para determinar el propietario de los ficheros recién creados, comprobar la máscara de permisos de acceso a ficheros y los permisos para enviar señales a otros procesos. Se utiliza también para acceder a archivos de otros usuarios.

- Identificador de grupo (GID): el identificador de grupo primario del grupo del usuario que lanza el proceso

- Identificador de grupo efectivo (EGID): puede ser distinto del de grupo, p.ej. en los programas que poseen el bit *setgid*

Tecleemos en el shell:

*ls -l /etc/passwd* y *ls -l /usr/bin/passwd*

*Normalmente el UID y el EUID coinciden, pero si un proceso ejecuta un programa que pertenece a otro usuario y que tiene el bit "s" (cambiar el identificador del usuario al ejecutar), el proceso cambia su EUID que toma el valor del UID del nuevo usuario. Es decir, a efectos de comprobación de permisos, tendrá los mismos permisos que tiene el usuario cuyo UID coincide con el EUID del proceso.*

# LECTURA DE ATRIBUTOS DE UN PROCESO

```
#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);
```

# LECTURA DE ATRIBUTOS DE UN PROCESO

*uid_t* ***getuid****(void);*

*uid_t* ***geteuid****(void);*

*gid_t* ***getgid****(void);*

*gid_t* ***getegid****(void);*

# JERARQUÍA DE PROCESOS

- El proceso de PID=1 es el programa *init*, que es el padre de todos los demás procesos

- Podemos ver la jerarquía de procesos con la orden *pstree*

# FUNCIONES DE TIEMPO Y CONTABILIDAD

- Tipos transcurridos:
  - Tiempo "de reloj de pared": tiempo transcurrido
  - Tiempo de CPU de usuario: cantidad de tiempo que utiliza el procesador para la ejecución de código en modo usuario (modo no núcleo)
  - Tiempo de CPU del núcleo: cantidad de tiempo utilizada en ejecutar código de núcleo
- La función *times* devuelve el tiempo "de reloj de pared" en ticks de reloj*:*

*#include <sys/times.h>* *POSIX*

*clock_t* ***times****(struct tms *buf);*

# FUNCIONES DE TIEMPO Y CONTABILIDAD

*#include <sys/time.h>* *no es POSIX*

*#include <sys/resource.h>*

*#include <unistd.h>*

*int **getrusage**(int who, struct rusage *rusage);*

- Da tiempo usado en código de usuario, tiempo usado en código del kernel, fallos de página

- *who*=proceso del que se quiere información

# CREACIÓN DE PROCESOS (I)

```
#include <unistd.h>
int system(const char *cmdstring);
```

- Crea un proceso que ejecuta un shell y le pasa la orden (*cmdstring*) para que la ejecute

- Devuelve el código retornado por la orden ejecutada en el shell, 127 si no pudo ejecutar el shell y -1 en caso de otro error

# CREACIÓN DE PROCESOS (II)

- Llamada al sistema ***fork:***

```
#include <unistd.h>
pid_t fork(void);
```

- Se crea un proceso idéntico al padre

- *fork* devuelve 0 al proceso hijo, y el PID del proceso creado al padre

# FUNCIONES DE TERMINACIÓN

- ***_exit*** vuelve al kernel inmediatamente. Definida por POSIX

*#include <unistd.h>*

*void **_exit**(int status);*

- ***exit*** realiza antes cierto "limpiado" (p.ej. terminar de escribir los buffers a disco). Es ANSI C

*#include <stdlib.h>*

*void **exit**(int status);*

- ***abort**():* el proceso se envía a sí mismo la señal SIGABRT. Termina y produce un *core dump*

# ESPERA POR UN PROCESO HIJO

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int options);
```

- Suspende al proceso que la ejecuta hasta que alguno de sus hijos termina
- Si algún hijo ha terminado se devuelve el resultado inmediatamente
- El valor retornado por el proceso hijo puede deducirse de *statloc*

# EJEMPLO DE FORK Y WAIT

```
if ( fork()==0 )
{
    printf ("Yo soy tu hijo!!! \n");
    exit(1);
}
else
{
    int espera;
    wait(&espera);
}
```

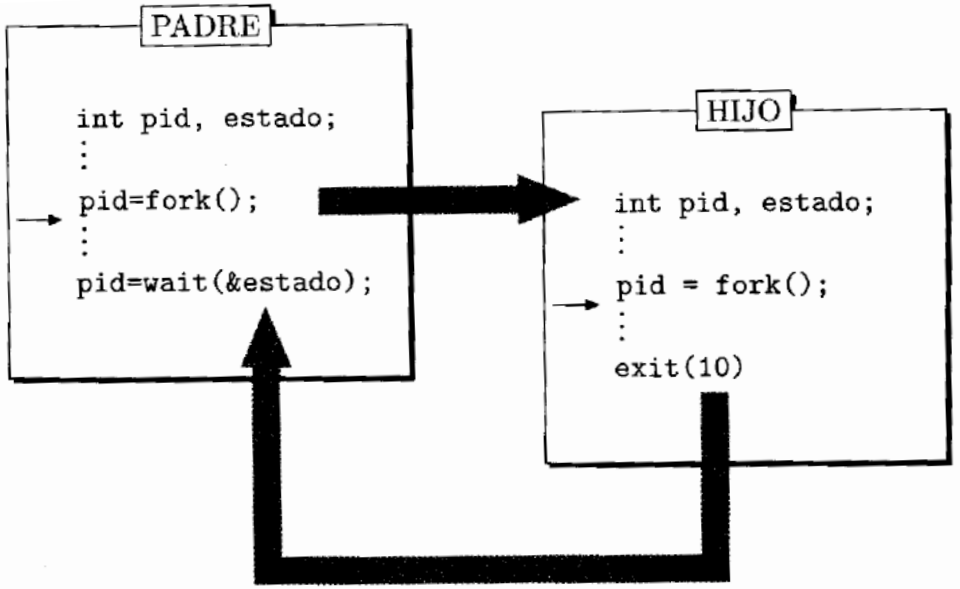



**Figura 6.3:** Sincronización entre los procesos padre e hijo.

# PROCESOS *ZOMBIES*

- Si un proceso padre no espera (*wait)* por la terminación de un proceso hijo, ¿qué pasa con éste? El proceso se queda zombi, o sea, queda con los recursos asignados, pero sin poder ejecutarse (no se le asigna CPU)
- El proceso hijo no puede desaparecer sin más, porque ha de comunicar su código de salida a alguien
- El proceso hijo habrá terminado (genocidio), pero permanecerá en el sistema (estado zombie).
- Cuando se haga el *wait* el proceso zombie se eliminará del sistema

# PROCESOS *ZOMBIES*

- ¿qué pasa si nunca hago el *wait?*
- Cuando el proceso padre termine, los hijos pasan a ser hijos del proceso *init*
- El proceso *init* elimina automáticamente los hijos zombies que tenga
- Y si el proceso padre no termina nunca (p.ej. un servidor)?
  - llamar *wait3, wait4* periódicamente (pueden ser no-bloqueantes)
  - manejar la señal SIGCHLD

# CREACIÓN DE PROCESOS (III)

- Para lanzar un programa, almacenado en un fichero (ejecutable)
- El proceso llamante es machacado por el programa que se ejecuta, el PID no cambia (el nuevo proceso absorbe al proceso llamador)
- Solo existe una llamada, pero la biblioteca estándar C tiene varias funciones, que se diferencian en el paso de parámetros al programa
- Ej:

```
char* tira [] = { "ls", "-l", "/usr/include", 0 };
...
execvp ( "ls", tira );
```

execvp busca el comando pasado como primer parámetro dentro del contenido de la variable PATH del environment del shell del intérprete de comandos. En este ejemplo lo ha encontrado en /bin y ejecuta /bin/ls

- Las llamadas retornan un valor no nulo si no se puede ejecutar el programa