TEMA 3 : PROCESOS

Sistemas Operativos: Programación de Sistemas

Oscar Déniz Suárez
Alexis Quesada Arencibia
Francisco J. Santana Pérez

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
Escuela Universitaria de Informática

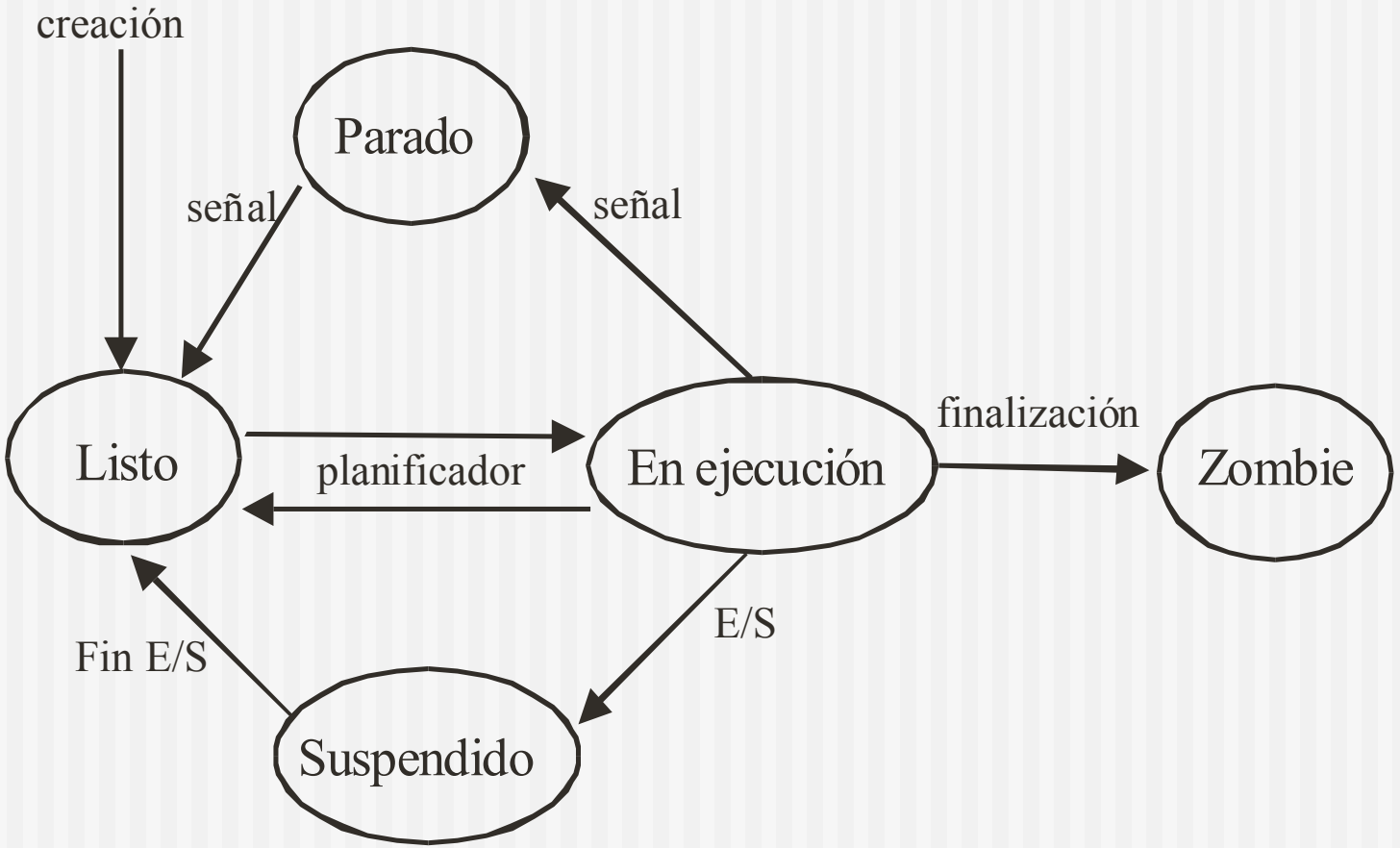
Curso 2006-07

(c) 2006 Alexis Quesada / Francisco J. Santana

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 los procesos

- 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

(c) 2006 Alexis Quesada / Francisco J. Santana

Identificadores de un 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 los atributos del proceso

#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);

1ª tarea práctica: escribir un programa que averigue qué pid tiene y la pid de su padre.

Si ejecutámos repetidamente el programa anterior, ¿qué observamos?

Lectura de los atributos del proceso

uid_t getuid(void);

uid_t geteuid(void);

gid_t getgid(void);

gid_t getegid(void);

2ª tarea práctica: escribir un programita que escriba el id usuario real y efectivo, y el grupo real y efectivo.

Luego cambiamos con chmod g+s nuestro programa y lo volvemos a ejecutar.

¿qué ocurre?

Modificación de atributos

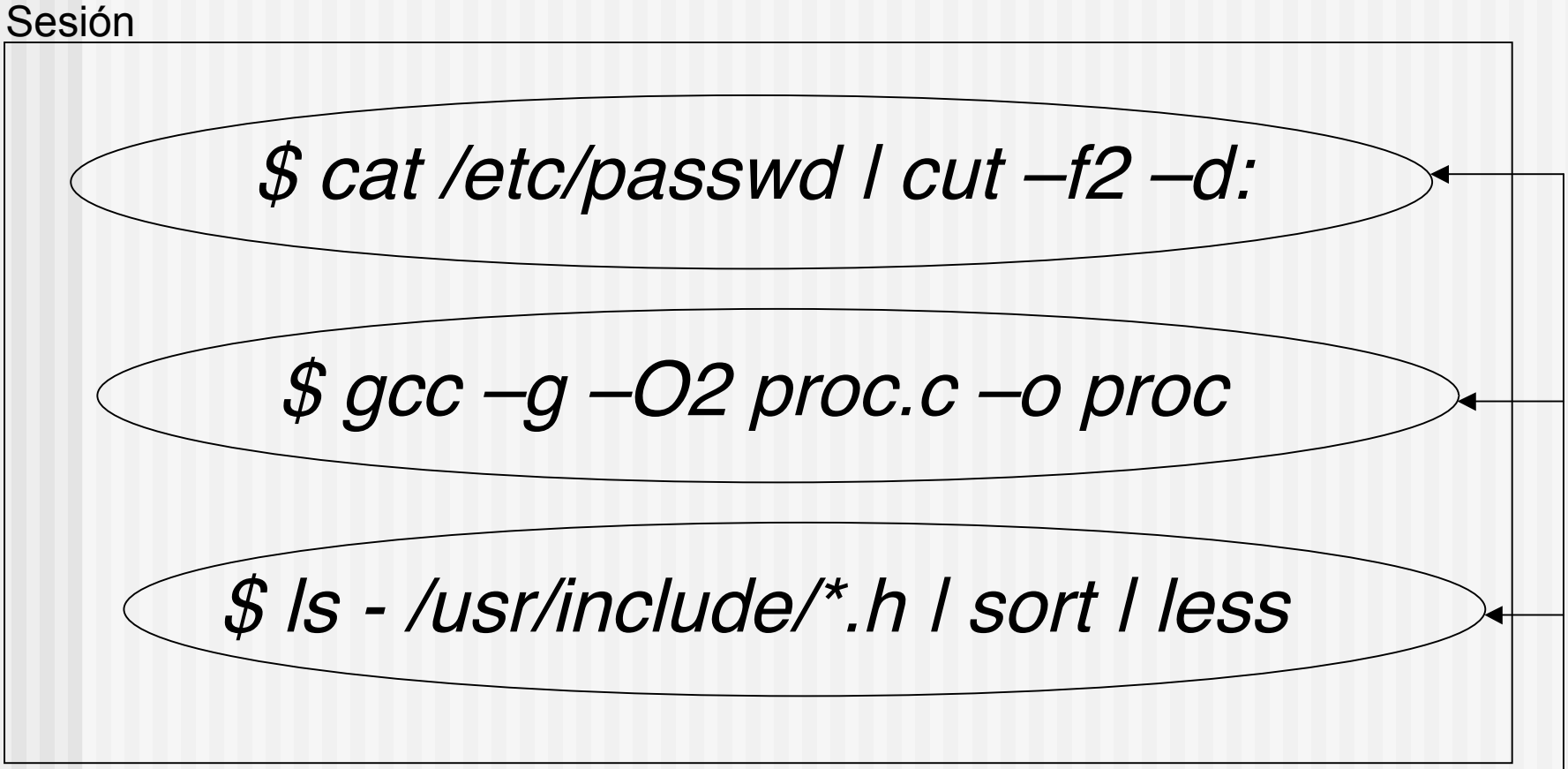
```
#include <unistd.h>
int setuid(uid_t uid);
int setreuid(uid_t ruid, uid_t euid);
int seteuid(uid_t euid);
int setgid(gid_t gid);
int setregid(gid_t rgid, gid_t egid);
int setegid(gid_t egid);
```

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 el comando *pstree*

Grupos y sesiones

Todos los grupos de procesos comparten el mismo PGID



Grupos de procesos

- Todo proceso forma parte de un grupo, y sus descendientes forman parte, en principio, del mismo grupo
- Un proceso puede crear un nuevo grupo y ser el *leader* del mismo
- Un grupo se identifica por el PID de su *leader*
- Se puede enviar señales a todos los procesos miembros de un grupo

Grupos de procesos

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
int setpgrp(void);
pid_t getpgrp(void);
```

Tiempos

- 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);*

Información de 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

La función *system*

#include <unistd.h>

int system(const char *cmdstring);

- Crea un proceso que ejecuta un shell y le pasa el comando para que lo ejecute
- Devuelve el código retornado por el comando de shell, 127 si no pudo ejecutar el shell y -1 en caso de otro error

Ejercicio Práctico

- Crear un programita que utilice *times* para visualizar los distintos tiempos transcurridos durante la ejecución del mismo.
- Utilizar bucles y llamadas a funciones para ir analizando los distintos tiempos.
- Utilizar una llamada a *system* y analizar tiempos tras su ejecución

Creación de procesos. *fork*

- 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 el 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 tonta;
    wait(&tonta);
}
```

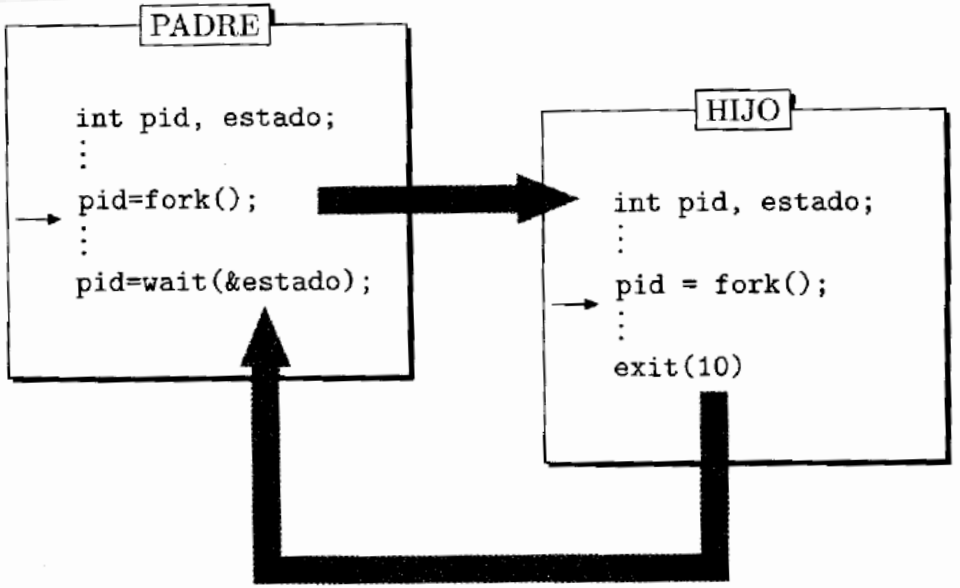


Figura 6.3: Sincronización entre los procesos padre e hijo.

Ejercicio Práctico

- Crear un programita que cree tres procesos hijos, en el que cada uno de ellos hará un bucle para escribir en pantalla un número del 1 al 100, en grupos de 10 (cada diez números), especificando en cada iteración el pid del hijo que la realiza.
- Usaremos RUSAGE_SELF y RUSAGE_CHILDREN (llamada al sistema *getrusage*) para analizar el tiempo que utilizan los hijos para realizar las iteraciones
- El padre deberá esperar hasta que los hijos finalicen para acabar, y dará un mensaje de aviso.
- Los hijos terminarán con funciones distintas de : _exit, exit y abort.

Procesos zombie

- 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 zombie

- ¿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

Las llamadas *exec*

- 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 absorve 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 };
...
execv ( "/bin/ls", tira );
```

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

Prioridad. *nice*

#include <unistd.h>

int nice(int inc);

- Por defecto, los procesos tienen un valor de *nice* 0
- *inc* > 0 => menor prioridad
- *inc* < 0 => mayor prioridad (solo superusuario)

Threads POSIX

- Un thread existe dentro de un proceso. Como los procesos, los threads se ejecutan concurrentemente
- Los threads de un proceso comparten su espacio de memoria, descriptores de ficheros, etc.
- Linux implementa el API estándar POSIX para threads, llamado *pthreads*
- Uso de pthreads:

#include <pthread.h>

Opción –lpthread del compilador

Threads POSIX.

- Para crear un thread necesitamos utilizar la llamada al sistema:

```
int  pthread_create (pthread_t  *thread, pthread_attr_t *attr,
                     void * (*start_routine)(void *), void * arg);
```

- **thread** es un apuntador a tipo de dato **pthread_t** que contiene el identificador del hilo de ejecución.
- **attr** es el apuntador a un tipo de dato **pthread_attr_t**, este indica que conjunto de atributos tendrá un hilo, entre otros: si un hilo puede obtener el estado de terminación de otro, que política de planificación tiene, que parámetros de planificación, etc. Para ver que tipos de atributos se tienen mirar pthread_attr_init(3). Se puede utilizar el valor NULL.
- **start_routine** es un apuntador a una función donde el hilo se ejecutará dicha función debe tener la siguiente firma:

```
void*
funcion_hilo(void *arg) {
  ...
}
```

- **arg** es el argumento que será pasado a la función que implementa el hilo.

Threads. POSIX.

- Ejercicio Práctico:
  - Crearemos una pequeña aplicación que cree un hilo:
    - El hilo deberá, constantemente, escribir por pantalla “hilo”.
    - El proceso que crea el hilo (padre), deberá escribir, constantemente, “padre”.