Programación de Sistemas

Ricardo Sanz

UPM-ASLab

Curso 2005-2006

Programas y Procesos

Los entes que habitan en las computadoras

Ejecución de un programa

- Del **fichero ejecutable** al **proceso en ejecución**
- Fases:
  - Asignación de espacio de direcciones
  - Copia de la imagen en memoria
  - Creación del prefijo de segmento de programa
  - Inserción en la lista “ejecutables”

El arranque en C

- Una **rutina especial de arranque** es llamada por el kernel
- La rutina es insertada en el ejecutable en el proceso de enlazado (link)
- La rutina se encarga de tomar del kernel los parámetros y el entorno
- Despues llama a la función main:
  *int main(int argc, char *argv[]);*

Terminación de procesos

- Un proceso puede terminar de muchas formas:
  - Normalmente:
    - retornando de *main* (con *return* ó llegando al final)
    - llamando a *exit*
    - llamando a *_Exit*
  - Anormalmente:
    - llamando a *abort()*
    - terminado por una señal no atendida

Terminación de procesos

- *_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);
```

Invoca funciones registradas con *atexit()*

Ver también _Exit()

Terminación de procesos

- La función *exit()* y *return* pueden devolver un código de retorno
- El shell puede utilizar el código de retorno para determinar el resultado de la ejecución programa
- La forma de hacerlo depende del shell (en *bash* se usa la variable de entorno "?")

Código de retorno

```
% ls /
bin coda etc lib misc nfs proc sbin usr
% echo $?
0
% ls fichero_no_existente
ls: fichero_no_existente: No such file or directory
% echo $?
1
```

Código de retorno 0 = terminación correcta

Terminación de procesos

- Con ANSI C un proceso puede registrar hasta 32 funciones que se llamarán automáticamente por *exit*
- La función *atexit:*

```
#include <stdlib.h>
int atexit(void (*func) (void));
```

Argumentos de línea de comando

```
int main(int argc, char *argv[])
{
    int i;
    for (i=0; i<argc; i++)
        printf("argv[%d]: %s\n", i,
    argv[i]);
    exit(0);
}
```

Variables de entorno

- Además de los parámetros, a un programa que comienza se le pasa la lista de variables de entorno.
- La lista es heredada del programa lanzador mediante un *extern*
- Es un array de punteros a cadenas terminadas en nulo (\0)

  *extern char **environ;*
- Las cadenas son de la forma variable=valor

Variables de entorno

```
#include <stdio.h>
/* The ENVIRON variable contains the environment. */

extern char** environ;

int main ()
{
char** var;
for (var = environ; *var != NULL; ++var)
      printf ("%s\n", *var);
return 0;
}
```

- ANSI C define una función útil:
  *char *getenv(const char *name)*

Estructura en memoria de un programa C

Codificación prudente: *assert*

- *assert(condición)*
- Para obviarlos se puede compilar definiendo la macro NDEBUG (opción –DNDEBUG del compilador)

Fallos de las llamadas al sistema

- Las llamadas al sistema podrán fallar por:
  - Argumentos de la llamada inválidos
  - Recursos agotados (p.ej. no hay memoria o disco suficiente)
  - Acceso no permitido (p.ej. al intentar escribir un fichero para el que no tenemos permiso)
  - Fallo hardware
  - Interrupción por una señal

Servicios básicos

- Servicios básicos que proporciona el sistema operativo para:
  - Procesos
  - Señales
  - Comunicación y sincronización de procesos
  - Memoria
  - Entrada/salida, ficheros
  - Manejadores de dispositivos

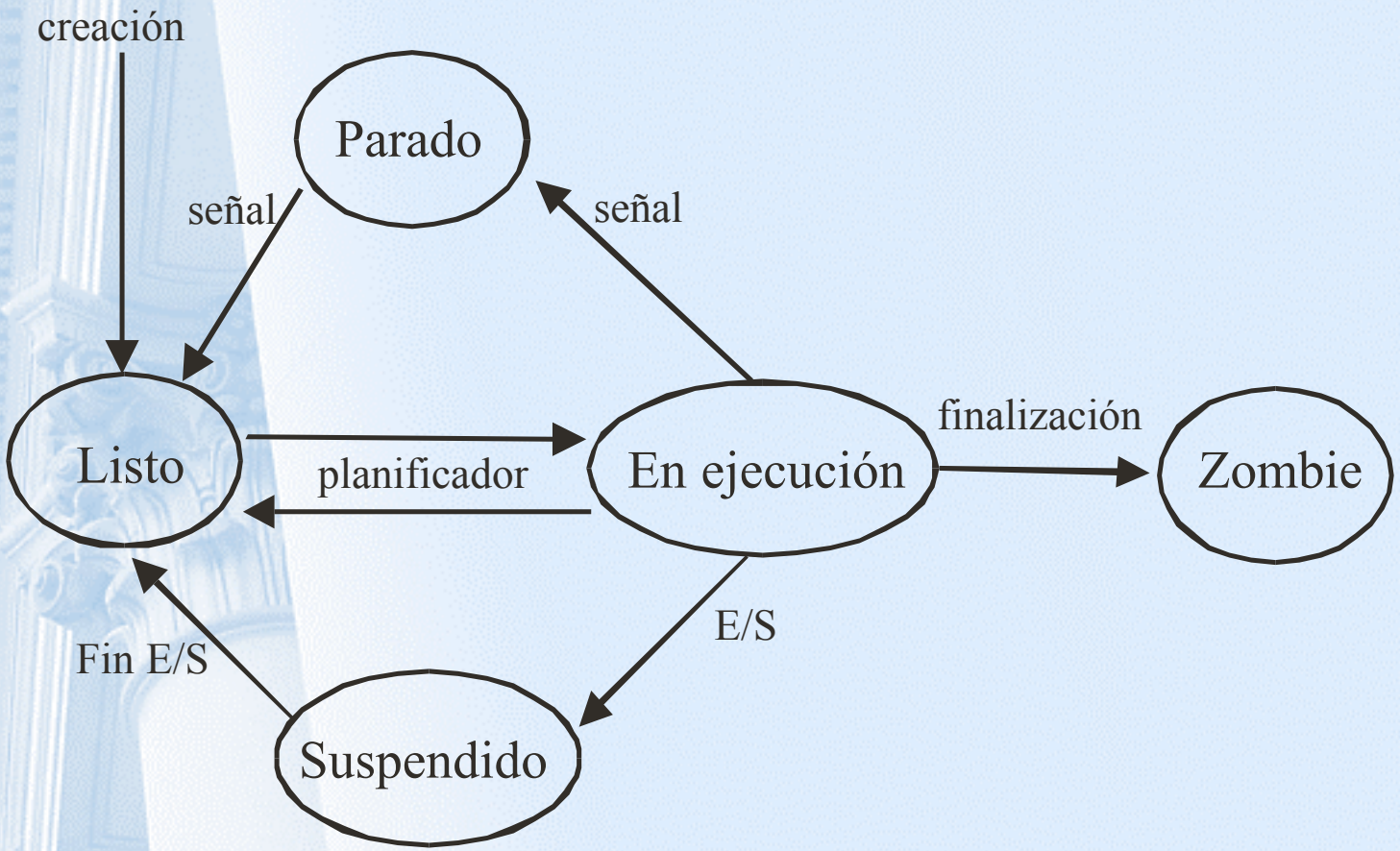
Servicios asociados a Procesos

Gestión de la ejecución de los programas

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
- PID 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

Identificadores de un proceso

- Identificador de usuario: el identificador del usuario que ha lanzado el programa
- Identificador de usuario efectivo: puede ser distinto del de usuario, p.ej en los programas que poseen el bit setuid
- Identificador de grupo: el identificador de grupo primario del grupo del usuario que lanza el proceso
- Identificador de grupo efectivo: puede ser distinto del de grupo, p.ej. en los programas que poseen el bit setgid
- Identificadores de grupo del usuario que lanza el proceso

Lectura de los atributos del proceso

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
int getgroups(int size, gid_t list[]);
```

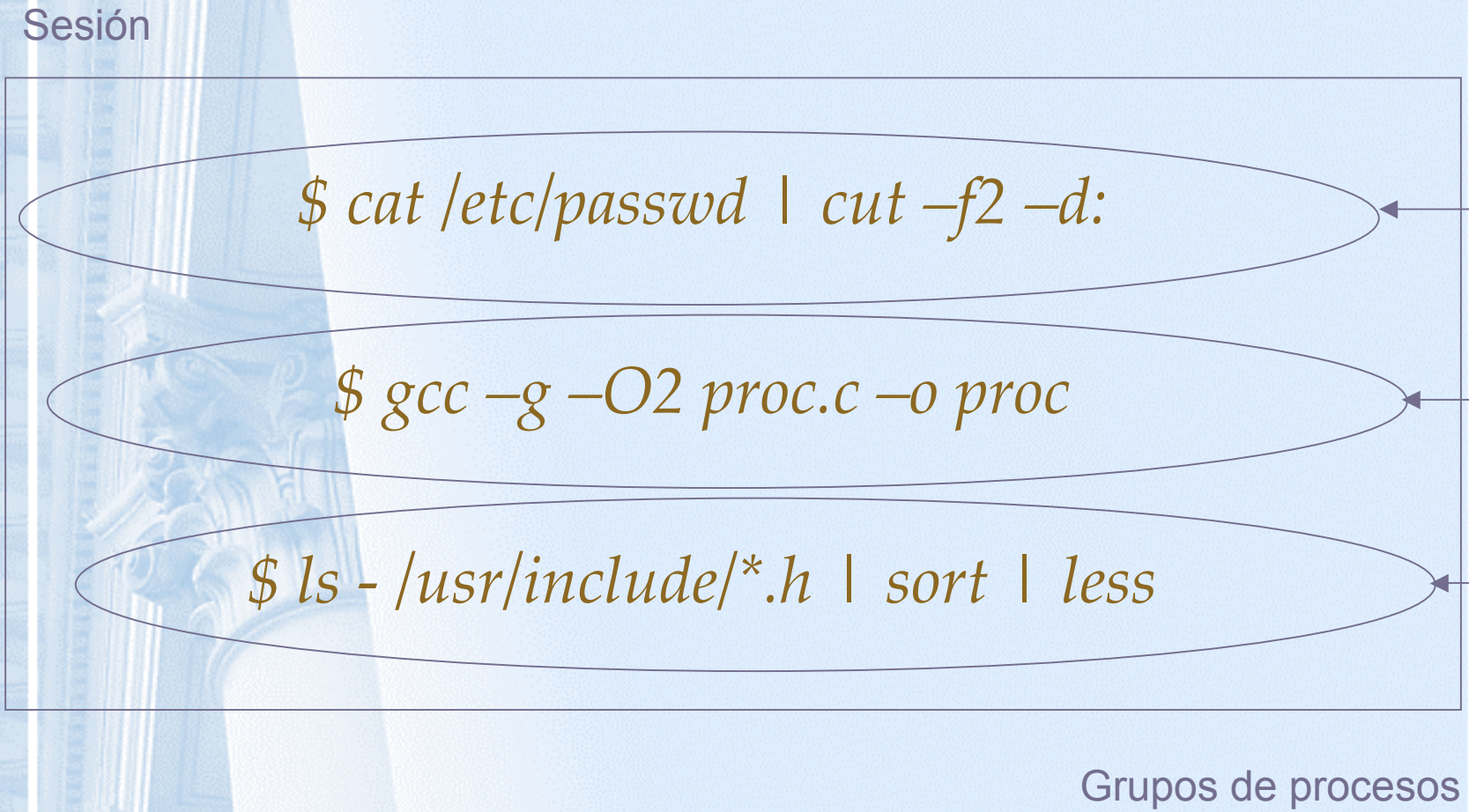
Modificación de atributos

```
#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



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 de CPU de usuario
  - Tiempo de CPU del núcleo

- La función *times* devuelve el tiempo “de reloj de pared” en ticks de reloj:

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

Información de contabilidad

```
#include <sys/time.h>
#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

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 mark;
        wait(&mark);
}
```

Procesos zombie

- Si un proceso padre no espera (*wait)* por la terminación de un proceso hijo, ¿qué pasa con éste?
- El proceso hijo no puede desaparecer sin más, porque ha de comunicar su código de salida a alguien
- El proceso hijo habrá terminado, 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
- El proceso llamante es machacado por el programa que se ejecuta, el PID no cambia
- 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

Procesos ligeros

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. Ejemplo

```
#include <pthread.h>
#include <stdio.h>

/* Prints x's to stderr. The parameter is unused. Does not return. */
void* print_xs (void* unused)
{
        while (1)
        fputc ('x', stderr);
        return NULL;
}
/* The main program. */
int main ()
{
        pthread_t thread_id;
        /* Create a new thread. The new thread will run the print_xs function. */
        pthread_create (&thread_id, NULL, &print_xs, NULL);
        /* Print o's continuously to stderr. */
        while (1)
        fputc ('o', stderr);
        return 0;
}
```

Señales

Señalización de eventos entre procesos

Señales

- Mecanismo para comunicar eventos a los procesos
- Cuando un proceso recibe una señal, la procesa inmediatamente
- Cuando un proceso recibe una señal puede:
  - Ignorar a la señal, cuando es inmune a la misma
  - Invocar la rutina de tratamiento por defecto
  - Invocar a una rutina de tratamiento propia

Señales

- Algunas señales importantes:
  - SIGTERM = Finalización controlada
  - SIGKILL = Finalización abrupta. No se puede ignorar
  - SIGINT = Interrupción. Se envía cuando se pulsa la tecla de interrupción (Ctrl+C)
  - SIGCLD = Terminación de algún proceso hijo. Se envía al proceso padre. Ignorada por defecto

Señales

- Señales disponibles para el programador: SIGUSR1 y SIGUSR2
- Para ver una cadena de texto sobre una señal:

```
#include <string.h>
#include <signal.h>
char *strsignal(int sig);
```

Envío de señales

- Para enviar una señal a un proceso o grupo de procesos:

```
#include <signal.h>
int kill(pid_t pid, int sig);
int killpg(int pgrp, int sig);
```

- Para enviar una señal a sí mismo:

```
#include <signal.h>
int raise(int sig);
```

Tratamiento de señales

```
#include <signal.h>
void (*signal (int sig, void (*action) ())) ();
```

- action puede ser:
  - SIG_DFL = acción por defecto
  - SIG_IGN = la señal se debe ignorar
  - Dirección a la rutina de tratamiento

Funciones para señales

- Una señal puede ser tratada o estar pendiente (en espera de ser tratada)
- Un proceso puede bloquear una señal; si se recibe quedará pendiente.
- La señal deja de estar pendiente cuando el proceso la desbloquea o cambia la rutina de tratamiento a la rutina por defecto
- *sigaction* permite instalar manejadores de señales
- *sigprocmask* permite especificar un conjunto de señales que queremos que el proceso bloquee
- *sigpending* devuelve el conjunto de señales que están bloqueadas y actualmente pendientes.

Conjuntos de señales

- Para cierta funciones (p.ej. *sigaction)* necesitamos especificar un conjunto de señales:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigmember(const sigset_t *set, int signo);
```

Manejador de señales

- *sigaction* es más nueva que *signal*, y es definida por POSIX. Permite ver o modificar (o ambas cosas) la acción asociada a una señal

```
#include <signal.h>
struct sigaction {
        void (*sa_handler)(int); /* dir. del manejador de señal o SIG_IGN o SIG_DFL */
        sigset_t sa_mask;               /* señales adicionales a bloquear cuando se maneje
            ésta*/
        int sa_flags;
        void (*sa_restorer)(void);      /* obsoleto */
}

int sigaction(int signum, const struct sigaction *act,struct sigaction
    *oldact);
```

Gestión del Tiempo

Control del tiempo en la ejecución de programas

Espera

```
#include <unistd.h>
int pause(void);
```

- *pause* deja al proceso suspendido en espera por una señal (cualquiera)
- Para esperar por una señal concreta:

```
#include <unistd.h>
int sigsuspend(const sigset_t *mask);
```

Funciones de tiempo

```
#include <sys/time.h>
struct itimerval {
        struct timeval it_interval; /* next value */
        struct timeval it_value; /* current value */
};
struct timeval {
        long tv_sec; /* seconds */
        long tv_usec; /* microseconds */
};
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
struct itimerval *ovalue);
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
unsigned int sleep(unsigned int seconds);
void usleep(unsigned long usec);
```

Funciones de tiempo

- Cada proceso puede tener varios temporizadores; se ponen con *setitimer*
- *alarm* genera una señal SIGALRM tras un número de segundos. La acción por defecto de SIGALRM es eliminar el proceso
- *sleep* y *usleep* suspenden al proceso un cierto número de segundos o microsegundos