Sistemas
Operativos

Procesos: Llamadas al Sistema

Procesos: Llamadas al Sistema

- Recuerda: main, imagen de un proceso
- Llamadas fork() y exec()
- Terminación de procesos: llamadas exit() y wait()
- Resumen de identificadores y marcas
- Intérpretes de comandos

[Ste05]: capítulo 8

Estructura de Programa en C

```
main(int argc, char  *argv[], char *envp[])
```

- Lista de argumentos: argv
- Lista de variables de entorno: envp

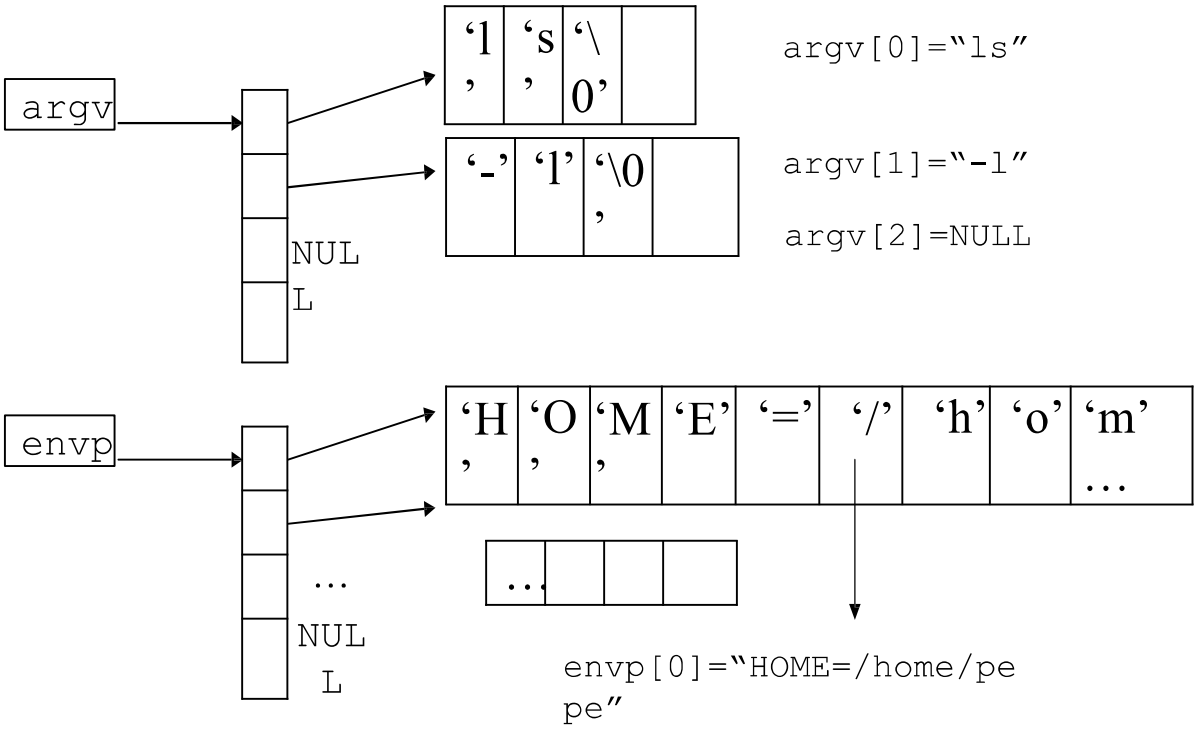
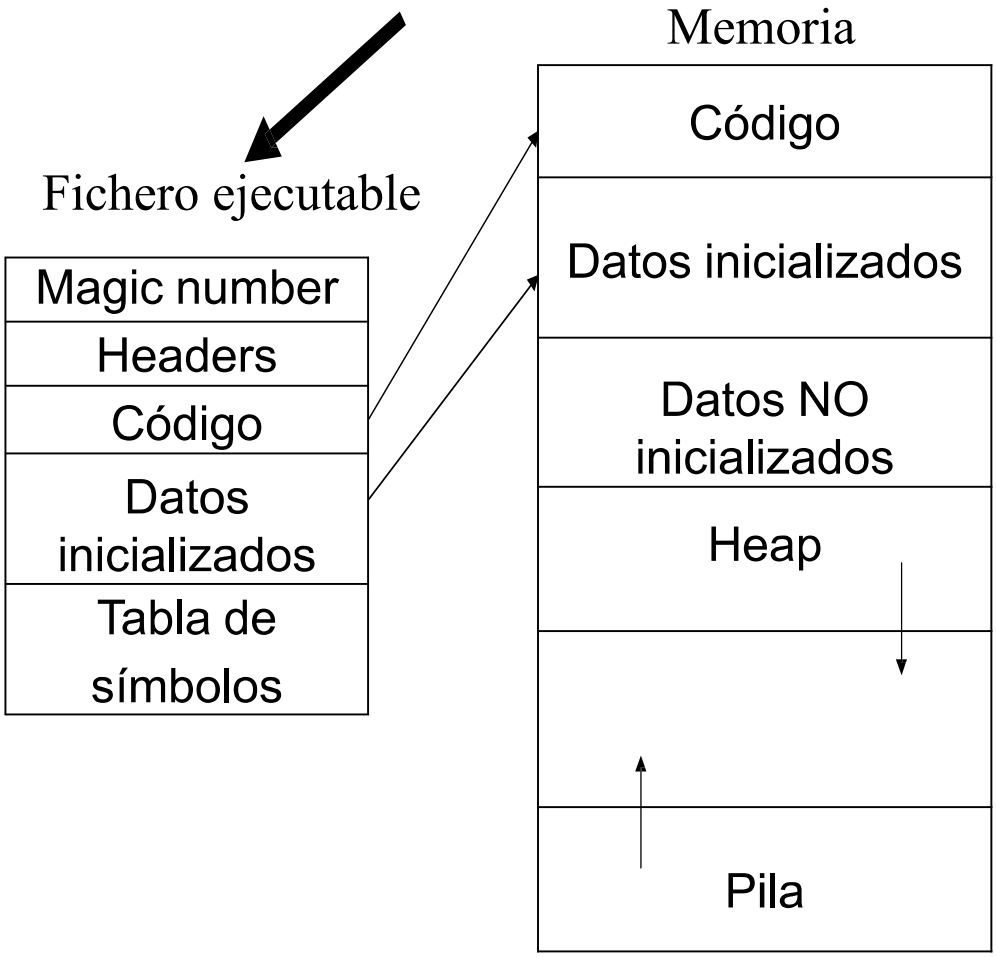


Imagen de un proceso

```
cc fichero.c -o fichero
```



UNIX: Llamadas asociadas

- Cada proceso en Unix se identifica por un pid (process identificator) número natural ≥ 0

  Algunos PIDs: 0 -> scheduler, 1-> init

- Gestión de procesos

– fork()	->	crear nuevo proceso (padre crea hijo)
– exit()	->	terminar (voluntariamente) proceso
– wait()	->	esperar terminación proceso (hijo)

- Carga y ejecución
  - exec() -> ejecutar programa
- Información
  - getpid() -> obtener PID de proceso (pid)
  - getppid() -> obtener PID del proceso padre (ppid)

fork()

- Crea un nuevo proceso
- La llamada `fork()` crea un duplicado (hijo) del proceso que la realiza (padre)
- Los dos procesos continúan ejecutando a partir de la llamada a la función `fork()`
- `fork()` devuelve el PID del hijo al proceso padre
- `fork()` devuelve 0 al proceso hijo
- `fork()` devuelve -1 si falla. Razones de fallo
  - Demasiados procesos en el sistema (causa exterior)
  - Demasiados procesos de usuario (CHILD_MAX, <limits.h>, 25 en hendrix)

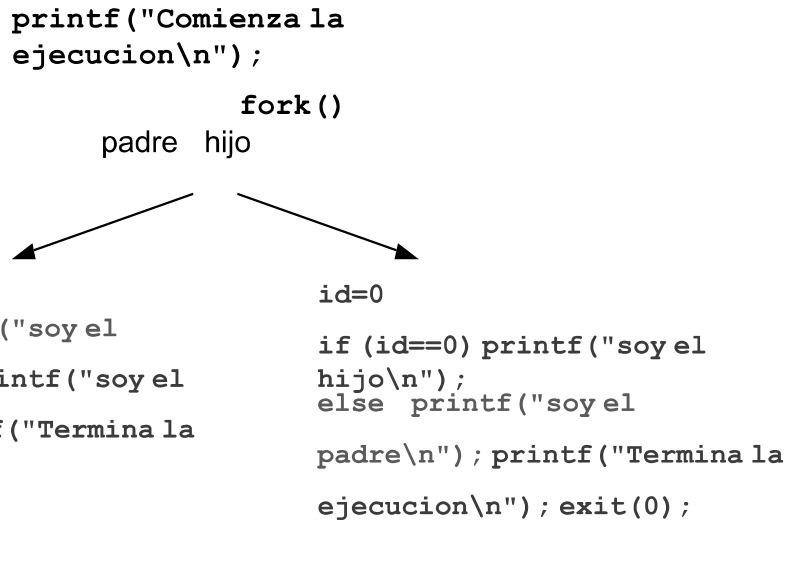
```
#include <unistd.h>



pid_t              /*  pid_t   es  un  int
fork(void);           /usr/include/sys/types.
                      h*/
```

Ejemplo **fork()**

```
main(){
   int  id;
   printf("Comienza  la
   ejecución\n");id=fork();
   if   (id==0)  printf("Soy  el
   hijo\n");else printf("Soy  el
   padre\n");
   /*   ejecutado   por ambos
   */printf("Termina la
   ejecución\n");exit(0);
}
```



exec()

- Pone en ejecución un programa (en fichero ejecutable)

```
main(){
   execl("/bin/ls","ls","
   -l",0);
   printf("Error\n");
}
```

- La llamada `exec()` sustituye la imagen del proceso que la realiza por la almacenada en un fichero ejecutable
  - En el ejemplo anterior, el `printf()` sólo se ejecuta si falla `exec()`
- La nueva imagen empieza a ejecutarse por la función `main()`
- La identidad del proceso no cambia. Sigue teniendo el mismo identificador, el mismo tiempo de CPU consumido, …
- Varios formatos (ver libro Stevens 2005):
  - `execl()`, **`execlp()`**, `execle()`
  - `execv()`, **`execvp()`**, **`execve`**`()`

exec()

- l -> argumentos del comando en lista
- v -> argumentos del comando en vector (como argv[])
- e -> nuevas variables de entorno en vector (como envp[])
- no e -> variables de entorno del usuario
- p -> filename (fichero ejecutable (usa variable PATH))
- no p -> pathname (trayectoria completa (no usa PATH))

#include <unistd.h>
int **execl**(const char *pathname,const char *arg0, ... /*
(char *)0 */); int **execv**(const char *pathname, char *const
argv []);
int **execle**(const char *pathname, const char *arg0, ...
/* (char *)0, char *const envp[] */);
int **execve**(const char *pathname, char *const argv[], char *const
envp []); int **execlp**(const char *filename, const char *arg0,...
/*(char *)0 */);
int **execvp**(const char *filename, char *const argv []);
Devuelven: –1 si hay error, nada si hay éxito (lógico)

wait()

#include
<sys/wait.h>
pid_t wait(int
*p_estado)

- Bloquea un proceso hasta que termine un hijo suyo (uno cualquiera si hay varios).
- El resultado de ejecutar wait() puede ser:
  - Bloqueo del proceso padre si todos sus hijos siguen ejecutandose
  - Retorna inmediatamente con el estado de terminación de un hijo (si esa información esta disponible)
  - Retorna inmediatamente -1 (error) si el proceso padre no tiene hijos
- waitpid(pid_hijo,*p_estado,options) -> Igual que wait, pero esperando a un hijo particular
  - Permite opciones adicionales para controlar bloqueos

Terminación de procesos

Voluntaria: `exit(0..255)`	Involuntaria: llega señal

- En ambos casos, el padre puede recibir información sobre la causa de la muerte del hijo mediante `wait()`

```
void
exit(estado)
int    estado
```

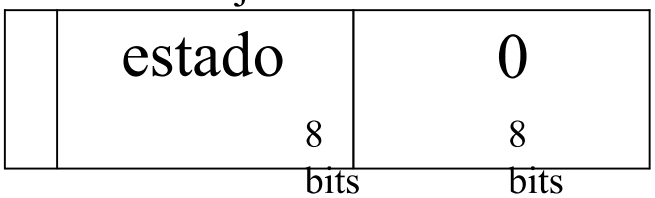
- No devuelve control nunca
- Estado: 0..255 (0: terminación normal)

```
int
wait(p_estado)
int*p_estado
```

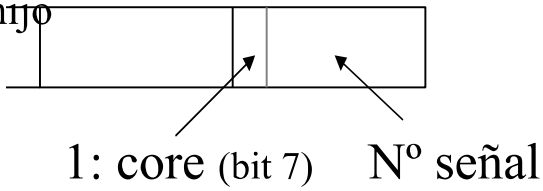
- Devuelve PID del hijo terminado
- Devuelve -1 en caso de ERROR

`*p_estado:`

Terminación voluntaria del hijo



Terminación involuntaria del hijo

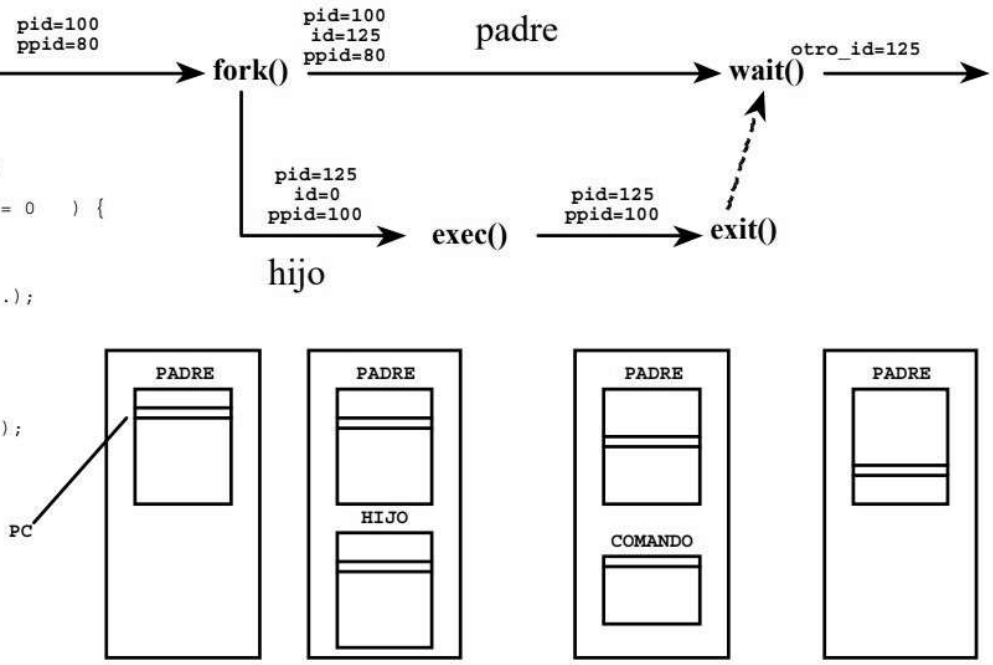


Información de hijo a padre

- Caso más frecuente
  - Proceso hijo termina y padre está esperando en wait()
    - Padre recibe información sobre la causa de la muerte del hijo y continúa ejecutando
    - Hijo desaparece
- Casos especiales
  - Proceso termina y padre no está en wait()
    - Proceso Zombie hasta que padre ejecute wait() o termine
  - Padre termina dejando hijos activos
    - Hijos adoptados por init (pid=1)
  - Padre ejecuta wait() y no existen hijos
    - Wait devuelve -1 y errno=10 (ECHILD: “No children”)

`fork()`, `exec()`, `wait()`: Ejemplo

```
main(){
    int id, estado, otro_id;
    if (   ( id = fork() ) == 0   ) {
        ...
        exec( "comando", ...);
    }
    ...
    otro_id = wait( &estado );
    ...
}
```



Ejemplo: **ej701.c**

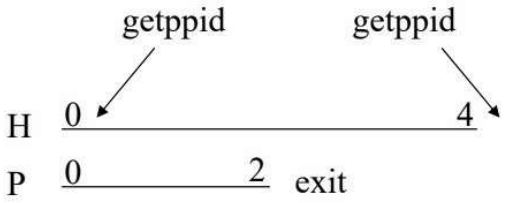
```
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
#include"error.h"

void main() {
  int pidh,err;
  printf("Inicio prueba\n");
  pidh=fork();
  if ( pidh == 0 ) {     /* hijo */
    printf("\n\tSoy el hijo: %d\n",getpid());
    printf("\n\tFork me devuelve: %d\n",pidh);
    exit( 0 );
  }
  /* padre */
  fprintf( stderr, "Antes de sleep\n");
  sleep( 1 );
  err=wait(NULL);
  if ( err == -1 ) syserr("wait");
  printf("\nSoy el padre: %d\n",getpid());
  printf("\nFork me devuelve: %d\n",pidh );
}
```

Ejemplo de hijo huérfano (**ej81.c**)

```
#include <errno.h>

int main() {
   int pid;
   switch(fork()) {
   case -1: perror("fork"); exit(1); /* error */
   case 0 :    /* hijo */
      pid=getppid();
      printf("pid padre antes = %d\n",pid);
      sleep(4); /* damos tiempo a que muera el padre.*/
      pid=getppid();
      printf("pid padre despues = %d\n",pid);
      exit(15);
   default:
      sleep(2);
      exit(0);
   }
}
```



Ejemplo de hijo zombie (**ej82.c**)

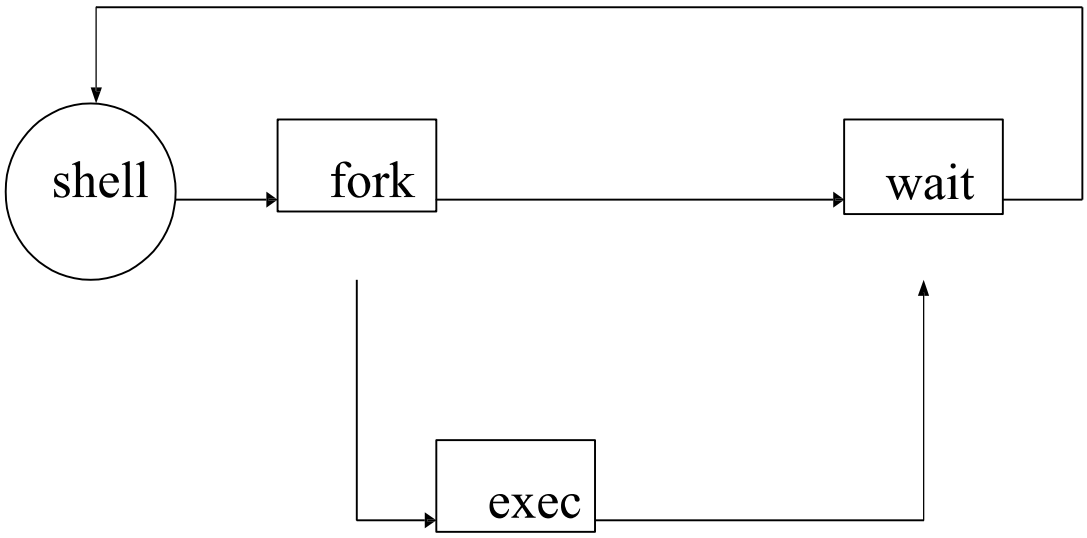
```
#include <errno.h>

int main() {
    int pid;
    int estado;
    switch(fork()) {
    case -1: perror("fork"); exit(1); /* error */
    case 0 :    /* hijo */
       sleep(2);
       printf("Soy el hijo %d y me muero...\n",getpid());
       exit(15);
    default: /*padre*/
       sleep(10); /* para que el hijo termine y quede zombie */
       if(-1==wait(&estado)){perror("wait"); exit(1);}
       printf("Estado hijo = %x\n", estado);
       exit(0);
    }
}
```

Arranque de UNIX

- ROM
- Carga del bootstrap
- Carga del resto del sistema
- Ejecuta kernel, gestor memoria, gestor ficheros
- exec(init) PID=1
- init ejecuta cada línea de /etc/inittab (fork y exec)
- Algunas líneas ejecutan /etc/getty (una por terminal)
- getty escribe “login:” y espera
  - Ejecuta /bin/login pasándole el username leído (exec)
- /bin/login escribe “password:” y espera
  - Comprueba password en /etc/passwd
  - Ejecuta el shell indicado en /etc/passwd (exec)
  - Shell es hijo de init, cuando acabe shell init lanzara otro getty

Estructura de shell



Algoritmo general de fork()

- Reservar PCB para el nuevo proceso
  - Comprobar que el número de procesos del usuario no excede el máximo
  - Buscar entrada libre en la tabla de procesos y un PID único
  - Marcar el estado del hijo como siendo creado
- Crear contexto del nuevo proceso
  - Copiar datos del padre al PCB del hijo
  - Inicializar los campos que difieran: tiempo, PID, ...
  - Reservar espacio en memoria y duplicar las zonas del padre
- *Modificar sistema de ficheros*
  - *Copiar la tabla de descriptores del padre sobre el hijo*
  - *Incrementar contadores de la tabla de ficheros abiertos*
- Devolver control
  - Devolver PID del hijo al proceso padre y cero al hijo como resultado de la función
  - Colocar a los dos procesos en estado preparado
  - Llamar al Scheduler

Algoritmo general de exec()

- Conseguir imagen del programa a ejecutar
  - Conseguir i-node del fichero ejecutable
  - Verificar número mágico y permiso de ejecución
  - Lectura de cabeceras. Comprobación de que es cargable
- Modificar contexto del proceso
  - Salvar temporalmente los parámetros de exec()
  - Liberar las regiones de memoria del proceso
  - Reservar espacio en memoria para las nuevas regiones
  - Cargar los contenidos del fichero ejecutable
  - Modificación de algunos campos del PCB:
    - Contador de programa, puntero de pila, ...
- Devolver control
  - Cargar variables de entorno y parámetros de la llamada exec() a la pila del proceso
  - Poner al proceso en estado preparado
  - Llamar al Scheduler

Resumen de Identificadores y Marcas (1)

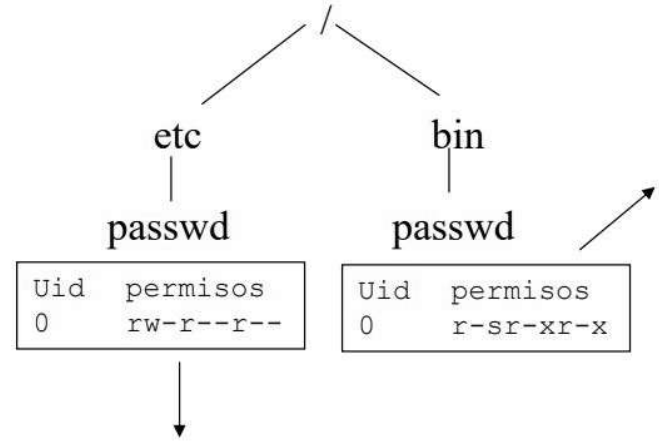
- Para cada usuario
  - UID (user id.): id. de usuario (nº natural)
    - UID=0 -> superusuario (root)
  - GID (group id.): id. de grupo al que pertenece (nº natural)
- Para cada fichero
  - Número de I-node: identificador numérico
  - UID del propietario
  - GID del propietario
  - Permisos de acceso (user-group-other)
  - Bits set-user-id, set-group-id: Permiten cambiar el usuario/grupo efectivo de un fichero.
  - Sticky bit: Permite gestión adecuada de ficheros de distintos usuarios en directorios compartidos (por ejemplo /tmp)

Resumen de Identificadores y Marcas (2)

- Para cada proceso
  - PID: identificador de proceso - getpid()
  - Process Group ID: Id. de grupo de procesos getpgrp()
    - setpgrp() PGID=PID (proceso líder)
    - setpgid(pid, pgrp)
  - Real User ID: UID del usuario que ha ejecutado el proceso getuid()
  - Real Group ID: GID del mismo usuario getgid()
  - Effective User ID: al ejecutar un programa, geteuid()
    SI (set-user-id==1 en el fichero ejecutable)
    EUID = UID del propietario del fichero (Ej: passwd)
    SINO EUID = RUID.
  - Effective Group ID: igual que EUID aplicado a grupo getegid()

Resumen de Identificadores y Marcas (3)

- Ejemplo de uso del bit set-user-id



- Nadie puede escribir sobre él, podría modificar los passwords
- Cualquiera puede leerlo, están encriptados

- Cualquiera puede ejecutarlo, sirve para cambiar el password
- Bit SETUID=1: el proceso que lo ejecuta pasa a tener EUID=0 (superusuario)
- Puede hacer cualquier cosa con /etc/passwd, puede modificarlo
- Durante un momento, el usuario que ejecuta /bin/passwd es superusuario:
  - No puede hacer nada que no haga /bin/passwd
  - No puede modificar /bin/passwd para que haga otras cosas

Herencia en fork()

- UID, GID, EUID, EGID, PGID
- Entorno, variables
  - cwd, umask, …
- *Comportamiento ante señales y mascara*
- *Tabla de descriptores de fichero*
- Otros
  - Segmentos de memoria compartida, límites de recursos del sistema, close-on-exec flag, terminal de control, …
- Diferencias entre padre e hijo
  - Valor devuelto por fork()
  - PID
  - Contadores de tiempo a cero en el hijo
  - *Ficheros bloqueados por el padre no lo están en el hijo*
  - *No se heredan las señales pendientes (ej. Alarmas)*

Herencia en exec()

- PID, PPID, PGID
- UID y GID (reales)
- Variables de entorno (salvo en execle y execve)
- *Señales pendientes*
- *Tabla de descriptores de fichero (salvo bit close-on- exec)*
- Otros
  - Bloqueo de ficheros, tiempo de ejecución acumulado, …
- En un exec cambia:
  - EUI, EGID (bits set-userID, set-groupID)
  - *Descriptores de fichero (bit close-on exec)*
  - *Comportamiento ante señales (no captura)*
  - Variables de entorno (en execle y execve)

Intérprete de comandos: **ej71.c**

```
#include <stdio.h> <stdlib.h> <unistd.h>
<sys/wait.h> #include "error.h"

int main(int argc,char *argv[]) { //comando sin
   parámetros switch ( fork() ){
   case -1:    /* error */
     fprintf( stderr, "no se puede crear proceso
     nuevo\n" ); syserr( "fork" );
   case 0:     /* hijo */
     execlp(argv[1],
     argv[1], 0);
     printf( "No se puede ejecutar %s\n",
     argv[1]); syserr( "execlp" );
   default:    /* padre
     */ if ( wait( NULL
     ) == -1 )
        syserr( "wait" ); /* trat.
     erroneo? */ printf( "Ejecutado %s
     \n", argv[1] );
   }
}
```

Bucle de ejecución: **ej73.c (1)**

```
#include <stdio.h> <string.h> <stdlib.h> <unistd.h> <sys/wait.h>
#include "error.h"

#define BUFSIZE 1024
int main( ) {
   static char s[BUFSIZE];
   char *argt[BUFSIZE];
   int i;
   for(;;){      /* bucle infinito */
     fprintf( stderr, "\n_$ " );
     if (gets(s)==NULL) {
         printf("logout\n");
         exit(0);
     }
     argt[0] = strtok( s, " " );  /* argt[0] comando a ejecutar */
     if( 0 == strcmp( argt[0], "quit" )){ /*cierto si argt[0]="quit" */
         printf("logout\n");
         exit( 0 );
     }
     for( i = 1; (argt[i] = strtok( NULL, " \t" )) != NULL; i++ );
```

Bucle de ejecución: **ej73.c (2)**

```
      switch ( fork() ){
      case -1:    /* error */
          fprintf(stderr, "\nNo se puede crear proceso
          nuevo\n"); syserr( "fork" );

      case 0:     /* hijo */
          execvp( argt[0],
          &argt[0]);
          fprintf(stderr,"\nNo se puede ejecutar %s\n",
          argt[0]); syserr( "execvp" );

      default:    /* padre */
          if( wait( NULL) == -1) syserr("wait");

      }/*switch*/
    }/*for*/
}/*main*/
```

Ejecución asíncrona: **ej1001.c (1)**

```
/* ej1001.c
   * Shell elemental con bucle de lectura de comandos
     con   parametros
   * Uso: [arre|soo] [comando][lista parametros]
*/
#include
<stdio.h>
#include
<string.h>
#include
<stdlib.h>
#include
<unistd.h>
#include
<sys/wait.h>
#include
"error.h"

#define BUFSIZE 1024
#define TRUE 1
#define FALSE 0

int main() {
   static char
   s[BUFSIZE]; char
   *argt[BUFSIZE];
   int i, parate, pid;
```

Ejecución asíncrona: **ej1001.c (2)**

```
while(1){
    fprintf( stderr,
    "\n_$ " ); gets( s );
    argt[0] = strtok( s, " " );
    if( 0 == strcmp( argt[0],
      "quit" )){
      printf("logout\n");
      exit( 0 );
    }
    for( i = 1; (argt[i] = strtok( NULL, " \t" )) != NULL; i++ );

    if( 0 == strcmp( argt[0], "soo" ) )

      parate
    = TRUE;
    else
      if( 0 == strcmp( argt[0],
          "arre" ) ) parate = FALSE;
      else{
          printf( "\n Mande?" ); continue;
      }
```

Ejecución asíncrona: **ej1001.c (3)**

```
switch ( pid=fork() ){
  case -1:    /* error */
      printf(stderr,"\nNo se puede crear proceso
      nuevo\n"); syserr( "fork" );

  case 0:     /* hijo */
      if(!parate) sleep(3);/*para que se note
      mas*/ execvp( argt[1], &argt[1]);
      fprintf(stderr,"\nNo se puede ejecutar %s\n",
      argt[1] ); syserr( "execvp" );

  default:   /*
      padre */
      if(parate)
          while( pid != wait(
               NULL)) if( pid
               == -1 ){
                        fprintf(stderr,"\nMuy raro.
                        Bye\n"); exit( 1 );
               }
  }/*switch*/
 }/*while*/
}/*main*/
```