unistd.h

Introducción

- **unistd.h** es el header file que provee acceso al API POSIX de los sistemas operativos Linux/Unix.
  - **API**: Application Programming Interface
    `en.wikipedia.org/wiki/Application_programming_interface`
  - **POSIX**: Portable Operating System Interface (standard, portabilidad)
    `en.wikipedia.org/wiki/POSIX`

- Es parte de la **C POSIX library**, junto con assert.c, limits.h, stdarg.h, stdio.h, etc
  `en.wikipedia.org/wiki/C_POSIX_library`

- Es una interfaz para las **syscalls**
  `en.wikipedia.org/wiki/System_call`

- **Header file `/usr/include/asm/unistd_32.h`**

```
#ifndef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H
/* This file contains the system call numbers. */
#define __NR_restart_syscall      0
#define __NR_exit                 1
#define __NR_fork                 2
#define __NR_read                 3
#define __NR_write                4
#define __NR_open                 5
#define __NR_close                6
#define __NR_waitpid              7
#define __NR_creat                8
#define __NR_link                 9
#define __NR_unlink              10
#define __NR_execve              11
#define __NR_chdir               12
#define __NR_time                13
#define __NR_mknod               14
#define __NR_chmod               15
#define __NR_lchown              16
#define __NR_break               17
#define __NR_oldstat             18
#define __NR_lseek               19
#define __NR_getpid              20
. . .
```

Funciones

getpid, getppid

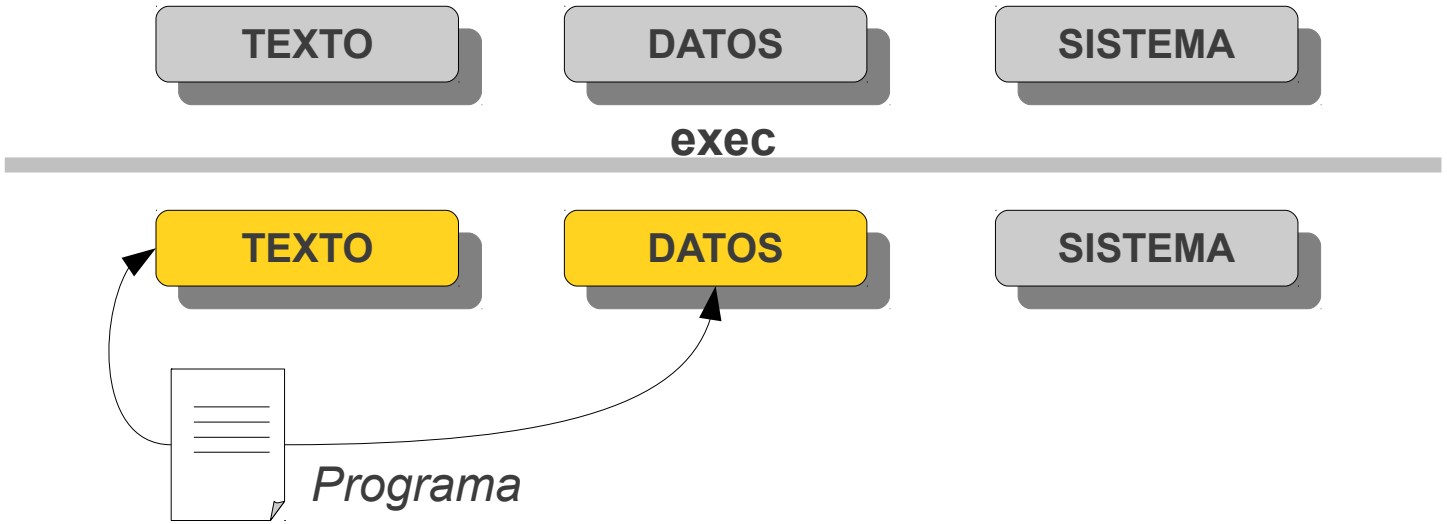
. Todo proceso en el sistema posee un identificador único PID (Process IDentificator)
. Todo proceso a excepción del proceso *init* tiene un proceso padre (Parent Process)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {

    printf("Mi pid es %d\n", getpid());
    printf("El pid de mi padre es %d\n", getppid());

    return 0;
}
```

Familia de funciones `exec`

• Concepto

- El proceso que invoca esta función es reemplazado por el recientemente creado
- Se trata de un reemplazo de procesos (imagen del proceso), el invocante se auto-termina



- El objetivo es ejecutar una imagen nueva del proceso

Familia de funciones `exec`

• **Exec** es una familia de funciones definidas en **`<unistd.h>`**

```
int execl  (const char *path, const char *arg, ...)
int execlp (const char *file, const char *arg, ...)
int execle (const char *path, const char *arg, char *const envp[])
int execv  (const char *path, char *const argv[])
int execve (const char *path, char *const argv[], char *const envp[])
int execvp (const char *file, char *const argv[])
```

Notas:

- Todas son front-ends de ***`execve`***
- `path` que indica la nueva imagen, `file` compone un path (si contiene / se usa como `path`, sino su prefijo es obtenido con PATH)
- La "`l`" viene de lista de argumentos `{arg1, arg2, ..., argn, NULL}` estos argumentos se transfieren al programa invocado
- "`v`" indica el uso de un vector de punteros a cadena, el *vector* (array) debe terminar en `NULL`
- "`e`" viene de *environment* (entorno) es una serie de variables con valores, sintaxis similar a la lista de argumentos

- **Familia de funciones `exec`**

Valor de retorno:

- En caso de éxito no hay valor de retorno ***¿por qué?***
- En caso de error retorna -1 y se establece *errno* para indicar el tipo de error

Razones

`[E2BIG]`: Se supera la cantidad de memoria (bytes) para los argumentos y el environment

`[EACCES]`: problemas de permisos

`[ENOENT]`: imagen no existe

etc.

Enlaces útiles

http://es.wikipedia.org/wiki/Errno.h

http://en.wikipedia.org/wiki/Perror

Exec: Ejemplos

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
   int ret;
   char *args[]={"ls", "-l", NULL};

   ret = execv("/bin/ls", args);

   if (ret == -1) {
      printf("errno = %d\n", errno);
      perror("execv");
      return -1;
   }

   printf("Codigo inalcanzable");
   return 0;
}
```

Exec: Ejemplos

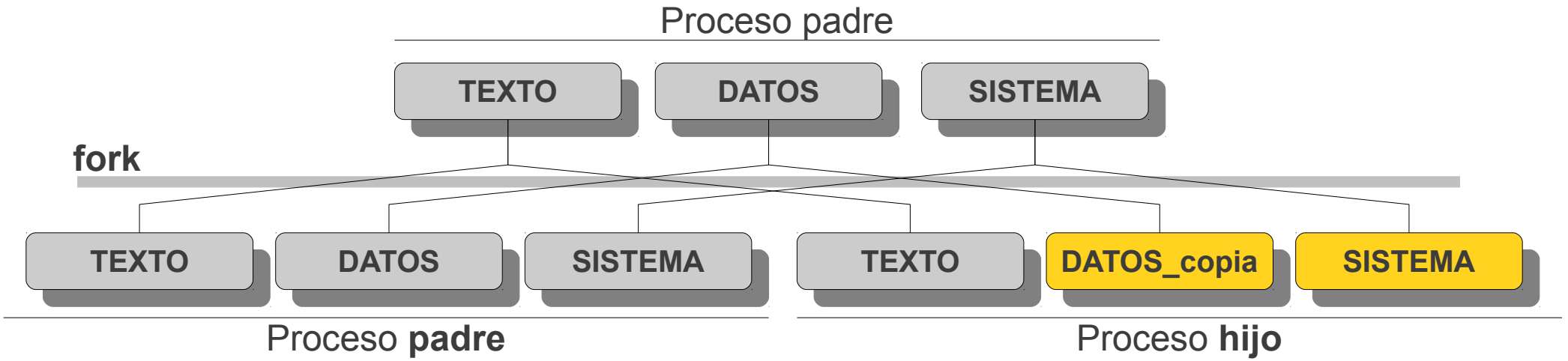
```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
   int ret;
   char *args[]={"sleep", "1000", NULL};
   char *env[]={"LOGNAME=gdm", "PWD=/opt", NULL};

   ret = execve("/bin/sleep", args, env);

   if (ret == -1) {
      printf("errno = %d\n", errno);
      perror("execve");
      return -1;
   }

   printf("Codigo inalcanzable");
   return 0;
}
```

```
$ cat /proc/PID/environ
LOGNAME=gdmPWD=/opt
```

- **`fork`**

• Crea un proceso nuevo (proceso hijo)

• Este proceso es una “copia” del proceso que invoca a fork



• Prototipo `pid_t fork (void)`

• Llamada exitosa

- Retorna el `PID` del proceso hijo al proceso padre
- Retorna 0 al proceso hijo

• En error retorna -1 y setea errno (`EAGAIN`, `ENOSYS`)

• El proceso hijo es una copia del proceso padre, excepto `PID` y `PPID`


```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
   pid_t pid_hijo = fork();
   int var = 5;

   switch(pid_hijo) {
     case 0: {   //proceso hijo
        var = 10;
        printf("Proceso hijo\n");
        printf("Hijo: PID=%d, PPID=%d - Variable=%d\n", getpid(), getppid(), var);
        break;
     }
     case -1: {
        printf("errno = %d\n", errno);
        perror("fork");
        break;
     }
     default: {  //proceso padre
        printf("Proceso padre\n");
        printf("Padre: PID=%d, PPID=%d - Variable=%d\n", getpid(), getppid(), var);
        break;
     }
  }
  return 0;
}
```

wait

- El proceso padre debe aguardar por la finalización del proceso hijo para recoger su exit status y evitar la creación de procesos zombies

- Prototipos

Almacena el exit status del proceso hijo Se analiza mediante macros

```
<sys/types.h>
<sys/wait.h>
pid_t wait (int *status)
pid_t waitpid (pid_t pid, int *status, int options)
```

- `wait` bloquea al proceso invocante hasta que el proceso hijo termina.
  Si el proceso hijo ya ha terminado retorna inmediatamente.
  Si el proceso padre tiene múltiples hijos la función retorna cuando uno de ellos termina.

- `waitpid` Posee opciones para bloquear al proceso invocante a la espera de un proceso en particular en lugar del primero.

- **Funciones**
  - **open, creat, read, write, close**

File descriptor es un entero positivo que se emplea en las llamadas al sistema para referenciar un acceso a archivo.
Actúan como índice en estructuras de datos internas en el sistema operativo donde se almacena la información de todos los archivos abiertos

```
<sys/types.h>
<sys/stat.h>
<fcntl.h>
int open(const char *pathname, int flags)
int open(const char *pathname, int flags, mode_t mode)
int creat(const char *pathname, mode_t mode)

<unistd.h>
ssize_t read(int fd, void *buf, size_t count)
ssize_t write(int fd, const void *buf, size_t count)
int close(int fd)
```

- **Funciones**

```
<sys/types.h>
<sys/stat.h>
<fcntl.h>
int open(const char *pathname, int flags)
int open(const char *pathname, int flags, mode_t mode)
int creat(const char *pathname, mode_t mode)
```

. **open** retorna un file descriptor a partir de un **pathname**
. En éxito, el file descriptor es el menor file descriptor no empleado por el proceso
. En fracaso retorna -1 y setea errno apropiadamente
. **flags** debe incluir uno de los siguienes accesos:
  O_RDONLY: sólo lectura
  O_WRONLY: sólo escritura
  O_RDWR: modo lectura/escritura
. **mode** especifica permisos a establecer para un archivo nuevo, vale si se establece O_CREAT en flags.
Los valores de **mode** posibles son: **S_IRUSR**, **S_IWUSR**... **S_IXOTH**
(pueden combinarse mediante bitwise-or)

creat(path, modo) ≡ **open**(path, O_CREAT | O_WRONLY | O_TRUNC, modo).

- **Funciones**

<unistd.h>

```
ssize_t read(int fd, void *buf, size_t count)
```

. Intenta leer **count** bytes desde el file descriptor **fd** (obtenido con **open**) y almacena los datos en el buffer **buf**.
. Retorna el número de bytes leídos o bien -1 en caso de error, seteando **errno**
. El número de bytes retornados puede ser:
  - menor que lo indicado en count
  - 0 significa que se llego al final del archivo
.

- **Funciones**

```
<unistd.h>
   ssize_t write(int fd, const void *buf, size_t count)
```

. Escribe hasta **count** bytes desde el bufer señalado por **buf** hacia el archivo referenciado por el file descriptor **fd** (obtenido con **open**).
. Retorna el número de bytes escritos o bien -1 en caso de error, seteando **errno**
. Si se retorna un entero menor que **count** estamos en presencia de algún error