LA BIBLIA DE LOS SISTEMAS OPERATIVOS

Autores:
Héctor Toral
Francisco Javier Pizarro
Javier Pardos

ÍNDICE

Toda llamada al sistema fallida devuelve -1

PROCESOS

```
#include <unistd.h>   // fork(), exec()
#include <sys/wait.h> // wait()
#include <stdlib.h>   // exit()
#include <stdio.h>    // printf()
```

“fork()”

Crea un nuevo proceso (hijo) duplicando el proceso de llamada. La ejecución de este continua desde donde se realizó la invocación a “fork()” que creó a este hijo en el proceso padre.

- Sintax

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

- Parámetros

- None

- Devuelve

- pid_t ⇒ El padre recibe el pid del hijo
- pid_t ⇒ El hijo recibe: 0

- Ejemplo

```
if ( fork() == 0 ) {...} else {...}
```

- Propiedades

- **no hay memoria compartida.**
- **hereda:**
  - UID, GID, EUID, EGID, PGID (llamadas al sistema para conseguir estos valores)
  - variables de entorno
  - comportamientos ante señales
  - **tabla descriptores de fichero**
- **No hereda:**
  - Señales pendientes(alarmas)
  - Ficheros bloqueados

- Casos

- Hijo muere ⇒ el padre lo recupera con wait, hasta entonces el hijo es un zombie, si finaliza sin haber recuperado al hijo, el hijo también finaliza
- Padre muere ⇒ el hijo es adoptado por init(pid=1, proceso init del S.0.)

IMPORTANTÍSIMO: PADRE E HIJO COMPARTEN CURSOR (esto existía antes del fork), SI EL PADRE LO MUEVE, AL HIJO TAMBIÉN SE LE MUEVE.

“exec()”

Ejecuta un programa/comando (si se ejecuta correctamente el exec, todo el código que sigue después de la forma que sea del exec es sustituido por el código del ejecutable que se va a ejecutar, en cambio si se ha producido un error seguiría con el código original del programa)

- Sintax

```
#include <unistd.h>
extern char **environ;

int execl(const char *pathname, const char *arg, ...
                        /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
                        /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...
                        /*, (char *) NULL, char *const envp[] */);

int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

- Propiedades

- Hereda:
  - 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-onexec),
- **- NO** hereda:
  - EUI, EGID (bits set-userID, set-groupID)
  - descriptores de fichero (bit close-on exec)
  - comportamiento ante señales (no captura)

- Ejemplos

exec**l**(): argumentos del comando en forma de lista.

```
// siempre se acaba con 0 o un puntero a 0
// como se ve en en la cabecera
execl("/bin/ls", "ls", "-l", "-a", 0);
```

exec**v**(): argumentos del comando en forma de vector.

```
char* cmd[4] = {"/bin/ps", "-lu", "pepito", NULL}
// con el & cogemos desde "/bin/ls" (cmd[0])
// hasta el NULL del final
execv(cmd[0], &cmd[0]);
```

int execle(const char* pathname, const char* arg0, ... /* (char*) 0 */, char* const envp[]);
int execve(const char* pathname, const char* argv[], char* const envp[]);

exec**le**() / exec**ve**(): exec usando nuevas variables de entorno (se pasan en el main como envp[] igual que argv[] y argc)

```
int main( int argc, char *argv[], char *envp[]) {
    char* cmd[4] = {"/bin/ps", "-lu", "pepito", NULL}
    // podemos pasarle cualquier char*[] como si hacemos lo siguiente:
    // char* entorno[] = {“hola”, “buenos”, “dias”}
    // y se lo pasamos, el uso es el correcto
    execve(cmd[0], &cmd[0], envp);
    execle(cmd[0], cmd[1], cmd[2], cmd[3], 0, envp);
}
```

"**cualquier forma de exec sin e**": usa las variables de entorno del usuario, no hace falta recibirlas como parámetros como si fuese argv[] ó argc (no esta definida envp[] en la cabecera del main)

int execvp(const char* filename, const char* argv[]);
int execlp(const char* filename, const char* argv0, ... /* (char*) 0 */);

exec**vp**() / exec**lp**(): usa la variable de entorno PATH para buscar el ejecutable de nuestros programas (**NO HACE FALTA PASAR LA RUTA COMPLETA DEL EJECUTABLE**)

```
execl("/bin/ls", "ls", "-l", 0);  ⇒ execlp("ls", "ls", "-l", 0);
```

"**cualquier forma de exec sin p**": siempre pasarle el PATH completos hasta el ejecutable en el primer parámetro de la función exec...()

"wait()"

Espera a que finalice un proceso hijo, devuelve el pid del hijo que ha finalizado, en p_estado devuelve el código de finalización de dicho hijo.

- Sintax

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

- Parámetros

- wstatus ⇒ código de salida del proceso hijo

- Devuelve

- pid_t ⇒ pid del hijo que ha terminado

- Ejemplo

```
wait(NULL); // espera a que el proceso hijo finalize
int p_estado;
wait(&p_estado);
// guarda en p_estado el código de salida al morir del hijo
```

"exit()"

Finaliza un programa/proceso, devuelve a su padre el código de finalización finalización.

Vacía los buffers.

- Sintax

```
#include <stdlib.h>
void exit(int status);
```

- Parámetros

- status ⇒ código de salida del proceso hijo

- Ejemplo

```
exit(0);//Finaliza todo correctamente
```

SEÑALES NO SEGURAS

#include<signal.h>
#define SIG_ERR (void (*)()) -1
#define SIG_DFL (void (*)()) 0
#define SIG_IGN (void (*)()) 1

SIGINT	2	terminación Ctrl C
SIGQUIT	3	terminación + core Ctrl Y
SIGKILL	9	terminación no ignorar/capturar
SIGPIPE	13	terminación escritura en pipe cerrada
SIGALRM	14	terminación se fija con alarm
SIGTERM	15	terminación como KILL pero si ignorar
SIGUSR[1-2]	16,17	terminación definidas por el programador
SIGCHLD	18	ignorada muerte del proceso HIJO

“signal()”

Reprograma el comportamiento ante la llegada de una señal, tras recibir la señal y hacer lo correspondiente **SE PIERDE ESTE REPROGRAMAMIENTO**
Devuelve el anterior comportamiento o SIG_ERR en caso de error

- Sintax

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- Parámetros

- signum ⇒ código de la alarma [SIGINT, SIGQUIT, ...]
- handler ⇒ función para sobreescribir el comportamiento por defecto

- Devuelve

- **sighandler_t:** devuelve el comportamiento por defecto

- Ejemplo

```
void fcaptura(int x) { // x contiene el valor de la señal que ha llamado a la función
    // REPROGRAMA DE NUEVO EL COMPORTAMIENTO DENTRO DE LA F.CAPTURA PARA NO PERDERLO
    signal(SIGALRM, fcaptura);
};

signal(SIGALRM, fcaptura); // reprograma la señal de alarma para ejecutar fcaptura
signal(SIGALRM, SIG_IGN);  // reprograma la señal de alarma para ser ignorada
signal(SIGALRM, SIG_DFL);  // reprograma la señal de alarma para tener el comportamiento
                           // default
```

“kill()”

Envía una señal a un proceso/ grupo de procesos

- Ejemplo

```
kill(pid, SIGUSR1);//manda la señal SIGUSR1 a pid
```

- Parámetros

- pid ⇒ identificador de proceso que va a recibir la señal “SIGUSR1”
- SIGUSR1 ⇒ Señal que se va a enviar al proceso “pid”

- Casos

- pid > 0 ⇒ destino = pid
- pid = 0 ⇒ destino = todos los procesos con process group id igual al del emisor (todos los procesos de su grupo)
- pid = -1 ⇒ procesos cuyo process group id igual al valor absoluto de pid
  a) si (emisor != superusuario) destino = todos los procesos con uid igual al euid del emisor
  b) si (emisor == superusuario) destino = todos los procesos (! PID=0, PID=1)

“alarm()”

Programa una señal de alarma para el propio programa para dentro de X segundos.
Si X == 0 cancela cualquier alarma pendiente.
Solo se puede tener una alarma pendiente.
Al ejecutarse devuelve los segundos que faltaban para ejecutar la alarma anterior, cancelando esta.

- Sintax

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

- Parámetros

- seconds ⇒ código de la alarma [SIGINT, SIGQUIT, ...]

- Devuelve

- **unsigned int:** devuelve el comportamiento por defecto

-Ejemplo

```
alarm(5);
```

“pause()”

Deja en estado bloqueado un proceso hasta que este recibe una señal que no sea ignorada por dicho proceso.

- Sintax

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

-Ejemplo

```
pause();
```

SEÑALES SEGURAS

```
#include <signal.h> // Señales seguras

int sigemptyset(sigset_t* set);
int sigfillset(sigset_t* set);
int sigaddset(sigset_t* set, int senial);
int sigdelset(sigset_t* set, int senial);
int sigismember(sigset_t* set, int senial);
int sigprocmask(int how, sigset_t* set, sigset* old_set);
int sigpending(sigset_t* set);
int sigsuspend(sigset* set);
int sigaction(int senial, struct sigaction* sigact, struct sigaction old_sigact)
```

```
// máscara de señales
// Si bit i = 0, señal i ∉ set
// Si bit i = 1, señal i ∈ set
sigset_t set;
```

Para el uso de señales seguras se debe crear una variable **sigset_t set** y después llamar a una de la siguientes funciones dependiendo de lo que se quiera conseguir:

```
// vacía el conjunto de señales, pone un 0 en los campos de la máscara
sigemptyset(&set)

// hace que toda señal del set, se active, todos tienen un 1 en la máscara
sigfillset(&set)
```

A partir de este estado (ninguna señal o todos las señales activada) se pueden usar las siguiente funciones:

```
//añadir señal a la máscara una por una, pone 1 en la máscara
sigaddset(&set, NUM_SEÑAL)

//quitar señal de la máscara una por una, pone 0 en la máscara
sigdelset(&set, NUM_SEÑAL)
```

También se puede usar la siguiente función para ver si una señal NUM_SEÑAL está activada dentro de la máscara:

```
sigismember(&set, NUM_SEÑAL) //comprobamos si NUM_SEÑAL está a 1 en la máscara
```

Para interactuar con el kernel del S.O podemos usar las siguiente funciones:

```
sigset old_set; //puntero a máscara de señales sin inicializar, como la var set
sigprocmask(HOW, &set, &old_set)//Siendo HOW uno de los siguiente valores:
                                //SIG_BLOCK: añade máscara del set al conjunto del S.O
                                //SIG_UNBLOCK: desbloquea las señales de la máscara del
                                //SO
                                //SIG_SETMASK: sustituye el set del SO por el pasado

// En set tenemos un conjunto de señales a 1 o 0, para bloquear, desbloquear o sustituir
// En old_set, podemos pasarle un puntero a un set, para guardar el set antiguo del S.O
// antes de realizar la función o en su defecto poner NULL y no guardamos nada

sigpending(&set) // Podemos consultar el estado de señales pendiente en el S.O
                 // Con la función nos guardamos en set, el conjunto de señales
pendientes

sigsuspend(&set) // Es como hacer un sigprocmask(..., ..., ...) + pause()
                 // Bloquea, desbloquea o actualiza el set de señales (set) y después
                //hace un pause() esperando una señal NO BLOQUEADA NI IGNORADA (todo
                //esto de FORMA ATÓMICA para evitar que una señal venga entre la
                //ejecución de estas dos instrucciones)
                //Al llegar la señal se va a la f.captura y restaura la máscara
                //anterior al hacer el sigsuspend()

stuct sigaction sigact;
stuct sigaction old_sigact;
sigaction(NUM_SEÑAL, &sigact, &old_sigact) //Es la versión segura del signal(...)
                                           //NUM_SEÑAL es el número de la señal a fijar
                                           //el comportamiento
                                           //sigact es el struct con el nuevo
                                           //comportamiento old_sigact es el struct con el
                                           //antiguo comportamiento si se desea guardar,
                                           //sino NULL

struct sigaction{
       void(*sa_handler)(); //SIG_IGN, SIG_DFL ó f.captura deseada
       sigset_t sa_mask;    //Máscara de señales a bloquear, se restaura al acabar la
                            //f.captura (dentro se ignoran y al acabar esta se vuelve a
                            //la mascara original)
       int sa_flag;         //Se pueden colocar una de la siguientes FLAGS:
                            // SA_RESTAR: reinicio de las system calls
                            //            (por defecto: NO REINICIO)
                            // SA_RESETHAND: reset a la f.captura original
                            //               (por defecto: NO RESET)
                            // SA_NODEFER: no bloqueo de la propia señal dentro de
                            //             f.captura
                            //             (por defecto: BLOQUEA SEÑAL DENTRO DE LA
                            //             F.CAPT)
}
```

EJEMPLO:

```
int main() {
    sigset_t mimask,oldmask;
    struct sigaction miaction;

    sigemptyset(&mimask);
    sigaddset(&mimask,SIGALRM);

    miaction.sa_handler=captura;
    sigemptyset(&miaction.sa_mask);
    sigaction(SIGALRM,&miaction,NULL);      //capturar señal SIGALRM

    printf("Señal SIGALRM capturada\n");

    sigprocmask(SIG_BLOCK, &mimask, &oldmask); //bloqueo SIGALRM antes de programar la
                                               //alarma
    printf("Señal SIGALRM bloqueada\n");

    /* región crítica */
    alarm(3);
    printf("Alarma programada\n");
    sigsuspend(&oldmask);
    /* fin region critica */

    printf("Fin programa\n");
    exit(0);
}

void captura(int n) {
    printf("Función de captura...%d\n",n);
}
```

FICHEROS/TUBERÍAS

PERMISOS

sigla: **U**ser **G**roup **O**thers
perm: rw- r-- r--
dec: 0 0 0
oct: 0X00 00X0 000X

X: en decimal

0 000 ---
1 001 --x
2 010 -w-
3 011 -wx
4 100 r--
5 101 r-x
6 110 rw-
7 111 rwx

"open()"

Abre el fichero especificado por pathname.
Si el fichero especificado no existe, es opcional (si O_CREAT está especificado en el modo) que se cree el fichero.

- Sintax

```
# include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- Parámetros

- pathname
  - ruta del fichero a abrir
- flags
  - flags para la apertura del fichero
  - e.g: O_RDONLY, O_WRONLY, O_RDWR
- mode
  - modo de apertura.
  - e.g: O_CREAT, O_APPEND, ...

- Devuelve

- int = fd ⇒ primer fd libre en TDF que apunta a TFA

- Ejemplo

```
int fd = open("prueba.txt",O_WRONLY|O_CREAT,0600);
```

"close()"

Cierra el descriptor de un fichero fd. Si se hace un close y quedan datos en el buffer se pierden (pueden quedar datos en stdout si se hace un printf sin \n al final y al hacer close(1) se perderían)

- Sintax

```
# include <unistd.h>
int close(int fd);
```

- Parámetros

- fd entero que identifica al descriptor de fichero que apunta a TDF

- Devuelve

- int = 0 ⇒ success

“creat()”

Crea un nuevo fichero.
Si ya existe el fichero pathname, reescribe este (trunca, pone todos los bytes del fichero 0).

- Sintax

```
# include <fcntl.h>
int creat(const char *pathname, mode_t mode);
```

- Parámetros

- pathname
  - ruta del fichero a crear
- mode
  - permisos con los que se creará el fichero (en octal).
  - e.g: 0777, 0700, ...

- Devuelve

- int = fd ⇒ primer fd libre en TDF que apunta a TFA

“read()”

Lee count bytes del fichero asociado al descriptor fd y los almacena en buf.
La lectura comienza en la posición indicada por el cursor (en TFA).
Se modifica la posición del cursor con el número de bytes leídos.

- Sintax

```
# include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- Parámetros

- fd entero que identifica al descriptor de fichero que apunta a TDF
- buf registro que almacena los bytes leídos
- count número de bytes a leer de fd

- Devuelve

- ssize_t = bytes | 0 ⇒ numero de bytes leidos | 0 si no hay más bytes para leer

“write()”

Escribe count bytes del buffer buf en el fichero referenciado por fd.
La escritura comienza en la posición indicada por el cursor (en TFA).
Se modifica la posición del cursor con el número de bytes realmente escritos.
Se actualiza el tamaño del fichero en i-nodo.

- Sintax

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

- Parámetros

- fd entero que identifica al descriptor de fichero que apunta a TDF
- buf registro del que se leen los bytes
- count número de bytes a leer de fd

- Devuelve

- ssize_t = bytes ⇒ número de bytes escritos

- Ejemplo read & write

```
char buffer[BUFSIZE];
while((bytes = read(0, &buffer, BUFSIZE)) > 0) {
    write(1, &buffer, bytes );
}
```

“lseek()”

Modifica la posición del cursor offset bytes.
Si offset > 0 hacia delante, si offset < 0 hacia detrás.
Se puede mover de forma relativa(pos. actual) o absoluta(principio / final).

- Sintax

```
# include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- Parámetros

- fd entero que identifica el descriptor del fichero
- offset número de bytes a mover del cursor `fd`
- whence tipo de movimiento [relativo | absoluto]
    - SEEK_SET, L_SET, 0 -> respecto del inicio del fichero
    - SEEK_CUR, L_CUR, 1 -> respecto a la posición inicial
    - SEEK_END, L_END, 2 -> respecto a la posición final

- Devuelve

- off_t = bytes -> offset resultante medido en bytes desde el comienzo del fichero

“stat()”

Recoge información del fichero asociado a la ruta pathname.

- Sintax

```
# include <sys/types.h>
# include <sys/stat.h>
# include <unistd.h>
int stat(const char *pathname, struct stat *statbuf);
```

- Parámetros

- pathname ruta del fichero
- statbuf registro donde almacenar la información recogida

- Devuelve

- int = 0 ⇒ success

“fstat()”

Recoge información del fichero asociado al descriptor de fichero `fd`

- Sintax

```
# include <sys/types.h>
# include <sys/stat.h>
# include <unistd.h>
int fstat(int fd, struct stat *statbuf);
```

- Parámetros

- fd descriptor del fichero
- statbuf registro donde almacenar la información recogida

- Devuelve

- int = 0 ⇒ success

Estructura de datos usada en stat() & fstat()

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
        precision for the following timestamp fields.
        For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Ejemplo stat() & fstat()

```
int main(int argc, char const *argv[]) {
    struct stat info_1;
    stat("fichero.txt", &info_1);
    printf("%d\n", info_1.st_size);

    struct stat info_2;
    int fd = open("fichero.txt", O_RDONLY);
    fstat(fd, &info_2);
    printf("%d\n", info_2.st_size);

    return 0;
}
```

“link()”

Crea para pathname_2 una nueva entrada en el directorio con el mismo i-nodo de pathname_1.
Número de links del i-nodo++.

- Sintax

```
# include <unistd.h>
int link(char *pathname_1, char *pathname_2);
```

- Parámetros

- pathname_1 fichero existente
- pathname_2 fichero nuevo

- Devuelve

- int = 0 ⇒ success

“unlink()”

Elimina la entrada en el directorio del fichero pathname:

```
[número de links]--
si [numero de links] == 0
        si [contador de uso en i-nodo] == 0
                borra fichero
        sino
                borrado en el ultimo close()
```

- Sintax

```
# include <unistd.h>
int unlink(char *pathname)
```

- Parámetros

- pathname fichero seleccionado

- Devuelve

- int = 0 ⇒ success

“pipe()”

Dado un vector de dos enteros() crea una tubería usándolo.
NOTA: Se puede leer y escribir de todo, en caso de que intentemos leer/escribir en una tubería que tiene un extremo cerrado da error.

- Sintax

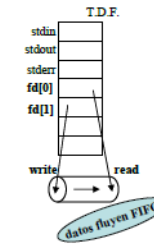
```
# include <unistd.h>
int pipe(int pipefd[2]);
```

- Parámetros

- pipefd vector en el que se crearán los descriptores de fichero para la pipe

- Devuelve

- int = 0 ⇒ success
- **pipefd[0] extremo de lectura**
- **pipefd[1] extremo de escritura**



“dup()”

IMPORTANTE: LAS MODIFICACIONES QUE SUFRA UNO DE LOS CURSORES, LAS SUFRIRÁN EL RESTO TAMBIÉN

Duplica el descriptor de fichero oldfd y lo pone en el primer hueco libre de la tabla de fds.

- Sintax

```
# include <unistd.h>
int dup(int oldfd);
```

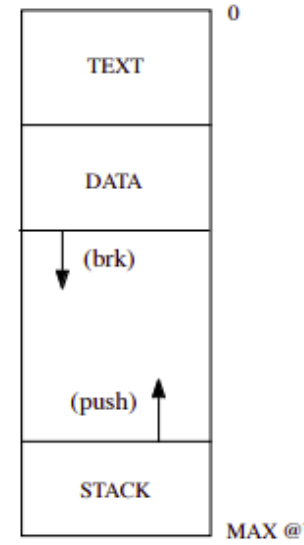
- Parámetros

- oldfd descriptor de fichero a duplicar

- Devuelve

- int = fd ⇒ primer fd libre en TDF que apunta a TFA

MEMORIA



“brk()” - No Usar

– Cambia límite superior del segmento de datos a end_ds
puede fallar por varias razones:
– Al ampliar la región se colisiona con otra
– Al ampliar la región se sobrepasa el espacio virtual máximo del proceso
– Faltan recursos:
· No hay espacio físico en memoria o en disco
(dependiendo de la implementación)

“sbrk()” - No Usar

– Suma increment bytes
al límite superior del segmento de datos
– Increment puede ser negativo
– devuelve el valor antiguo de end_ds

FUNCIONES DE LIBRERÍA

```
#include <stdlib.h>
#include <sys/mman.h>
```

“malloc()”

– Reserva espacio para al menos size bytes. No inicializa el espacio

- Sintax

```
#include <stdlib.h>
void *malloc(size_t size);
```

- Parámetros

- size ⇒ número de bytes reservados

- Devuelve

- void* ⇒ puntero a la zona reservada

- Ejemplo

```
integer *x, *y;
integer n = 4;
integer res = 0;
x = malloc(n*sizeof(int)) // reserva n*4 bytes
y = malloc(n*sizeof(int)) // reserva espacio para n int
// Al hacer malloc reservamos espacio PERO NO INICIALIZA MEM. RESERVADA
cargarVector(x, n);
cargarVector(y, n);
for(int i = 0; i<n; i++){
    res+=x[i]+y[i];
    // operamos con los datos apuntado por punteros, por ejemplo para
    // sumar las componentes.
}
```

“calloc()”

Reserva espacio para n elementos (nmemb) cada uno de size bytes.
Inicializa el espacio reservado con ceros.

- Sintax

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

- Parámetros

- nmemb ⇒ número de elementos de un array
- size ⇒ número de bytes que ocupa cada elemento del array

- Devuelve

- void* ⇒ puntero a la zona reservada

“free()”

- Sintax

```
#include <stdlib.h>
void free(void *ptr);
```

- Parámetros

- ptr ⇒ puntero a la zona a liberar

- Devuelve

- void* ptr (pero no es útil usar este puntero porque esta zona de memoria podría usarla otro proceso al ser liberada)

– Libera el espacio apuntado por ptr. **Este espacio no queda liberado para el Kernel**
– Hay una estructura de gestión de espacios propia de la librería
– No mezclar llamadas a brk() con funciones de librería.

“realloc()”

Cambia el tamaño del bloque apuntado por ptr al valor size.
Devuelve puntero al inicio de bloque, puede ser distinto del original.
Si ptr = NULL actúa como malloc().
Si size = cero actúa como free().

- Sintax

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- Parámetros

- ptr ⇒ puntero de la zona de memoria que se desea realojar
- size ⇒ si [0 , Null, void*] entonces se hace free()

- Devuelve

- void* ptr ⇒ puntero a la zona de mem que se ha modificado, puede ser distinto al original

“mmap()”

- Sintax

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

- Parámetros

- addr ⇒ dirección inicial de la nueva asignación. Si **addr == [0| Null]** se coloca donde puede (recomendable)
- length ⇒ longitud de la asignación. longitud > 0
- prot ⇒ protección de memoria deseada de la asignación (no debe entrar en conflicto con el modo de apertura del archivo).
    - PROT_EXEC ⇒ Las páginas pueden ser ejecutadas
    - PROT_READ ⇒ Las páginas pueden ser leídas
    - PROT_WRITE ⇒ Las páginas pueden ser escritas
    - PROT_NONE⇒ Las páginas no pueden ser accedidas
- flags ⇒ El argumento flags determina si las actualizaciones del mapeo son visibles para otros procesos que mapean la misma región, y si las actualizaciones son llevadas a el archivo subyacente.
    - MAP_SHARED ⇒ un store sobre la región = write sobre fichero
    - MAP_PRIVATE ⇒ un store provoca una copia privada el fichero nunca se modifica
    - MAP_FIXED ⇒ addr pasa de hint a obligación
- fd ⇒ descriptor de fichero. Se puede cerrar sin invalidar el mapeo.
- offset ⇒ define el número de bytes desde el que se comenzará el mapeo. Debe ser múltiplo de página

- Devuelve

- puntero a la zona mapeada

- Ejemplo

```
/* mreverse.c Muestra el contenido de un fichero al revés por la terminal.*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h> /* mmap() */
#include <fcntl.h>
#ifndef MAP_FILE
#define MAP_FILE 0
#endif
int main(int argc,char* argv[]) {
    int fdfnt;
    long fsize;
    char *src;
    fdfnt = open(argv[1], O_RDONLY);
    fsize=lseek(fdfnt,0L,2);
    src = mmap(0, fsize, PROT_READ, MAP_FILE | MAP_SHARED, fdfnt, 0);
    fsize--; // por el fin de fichero
    while(fsize >= 0){
        write(1, &src[fsize], 1);
        fsize--;
    }
}
```

THREADS

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

Hilos de usuario (User Level Thread, ULT)
– La gestión de hilos la lleva la propia aplicación (librería)
– El SO no da ningún soporte, no conoce los hilos
Hilos de sistema (Kernel level Thread, KLT)
– también llamados: kernel supported threads, lightweight processes
– El SO conoce y gestiona hilos, planifica a nivel de hilo
pthread_t id;
Crea un thread llamado id

ejemplo de función y argumentos:

```
typedef struct {
        int *vector;
        int *sol;
        int n;
} miarg;

void *funcion(void *arg) {
        miarg *p = (miarg *)arg;
        const int i = p->n;
        p->n++;
        p->sol[i] = 0;
        for(int j = MAX/Nt*i; j < MAX/Nt*(i+1);j++) {
                p->sol[i] += p->vector[j];
        }
        return NULL;
}

miarg args = {vec,sol,0};
```

pthread_create(&id, NULL, funcion, &args);
hace que el thread llamado id ejecute la función con los argumentos args
pthread_join(id, NULL);
espera a que el thread id finalice
pthread_mutex_t algo;
crea un mutex, se debe crear como variable global
pthread_mutex_init(&algo, NULL);
inicializa el mutex, en el main
pthread_mutex_lock(&algo);
bloquea el mutex(garantiza la exclusión mutua)
pthread_mutex_unlock(&algo);
desbloquea el mutex

C PARA TONTICOS

CONVERSIONES STRING ENTERO

```
atoi();  // pasa de char* a entero
itoa();  // lo contrario
```

PASAR POR REFERENCIA A UNA FUNCIÓN

```
#include <stdio.h>

void intercambia(int* x, int* y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int a = 1, b = 2;
  intercambia( &a, &b );
  printf( "\n\ta = %d\tb = %d\n\n", a, b );
}
```

STRTOK

Para ejecutar un comando que tengamos almacenado en una línea que está guardada en buffer

```
// buffer = almacena una línea que contiene un comando con hasta 100 par
char buffer[BUFSIZE];
char* aux[100];
// aux[0] = comando que tiene que buscar en el path
aux[0] = strtok(buffer, " ");
// actualiza el vector "aux" con los valores recogidos en las línea de buffer
for(int i = 1; (aux[i] = strtok(NULL, " \t")) != NULL; i++);
// ejecuta el programa
execvp(aux[0], &aux[0]);
```

Para leer de un fichero y usar una “función” gets
En este caso se van leyendo líneas y se escriben en fd hasta llegar a n líneas

```
if (aux < n) {
    líneas = strtok(&buffer, "\n");
    write(1,lineas,sizeof(lineas));
    if (lineas != NULL) write(1,"\n",1);
        aux++;
}

int bre = 0;
for(int i = 1; bre == 0; i++) {
    líneas = strtok(NULL, "\n");
    if (aux < n) {
        write(1,lineas,sizeof(lineas));
        if (lineas != NULL) write(1,"\n",1);
        else bre = 1;
        aux++;
    }
    if (lineas == NULL) bre = 1;
    write(fd,&buffer,nRead);
}
```

LLAMADAS PARA OBTENER INFORMACIÓN

PID	Process ID	getpid()
PGI	Process Group ID	getpgrp()
UID	Real User ID	getuid()
GID	Real Group ID	getgid()
EUID	Effective User ID	geteuid()
EGID	Effective Group ID	getegid()

setpgrp() PGID=PID (proceso líder)
setpgid(pid, pgrp)
SI (set-user-id == 1 en el fichero ejecutable)
EUID = UID del propietario del fichero
SINO EUID = UID