Fundamentos de Sistemas Operativos

Introducción al Lenguaje de Programación C III

Universidad de Valladolid

Departamento de Informática

Curso 2023/2024

Contenidos

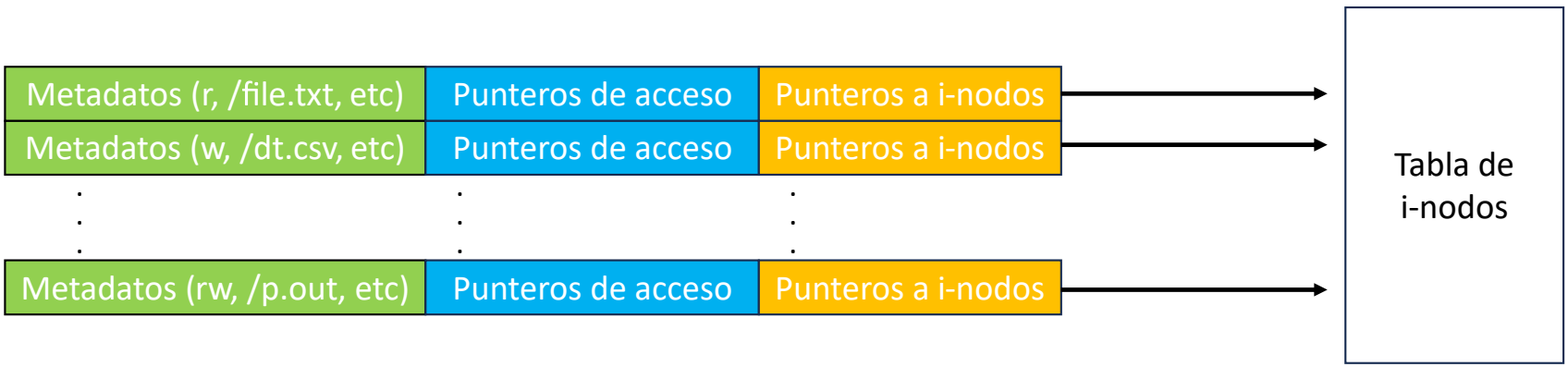
Descriptores de archivo

¿Qué es la Tabla de Archivos Abiertos? (*Open File Table*)

- Es una tabla mantenida en memoria por el *kernel* del sistema operativo que tiene una entrada por cada archivo abierto en el sistema operativo.
- Cada vez que se ejecuta con éxito la apertura de un archivo (*open*), se crea una nueva entrada en esta tabla.
- La tabla mantiene información de los punteros de lectura/escritura actuales, los permisos de acceso de cada proceso que puede acceder al archivo, la ubicación del archivo, sus metadatos y su entrada a la tabla de i-nodos del disco duro.

Descriptores de archivo

Tabla de Archivos Abiertos



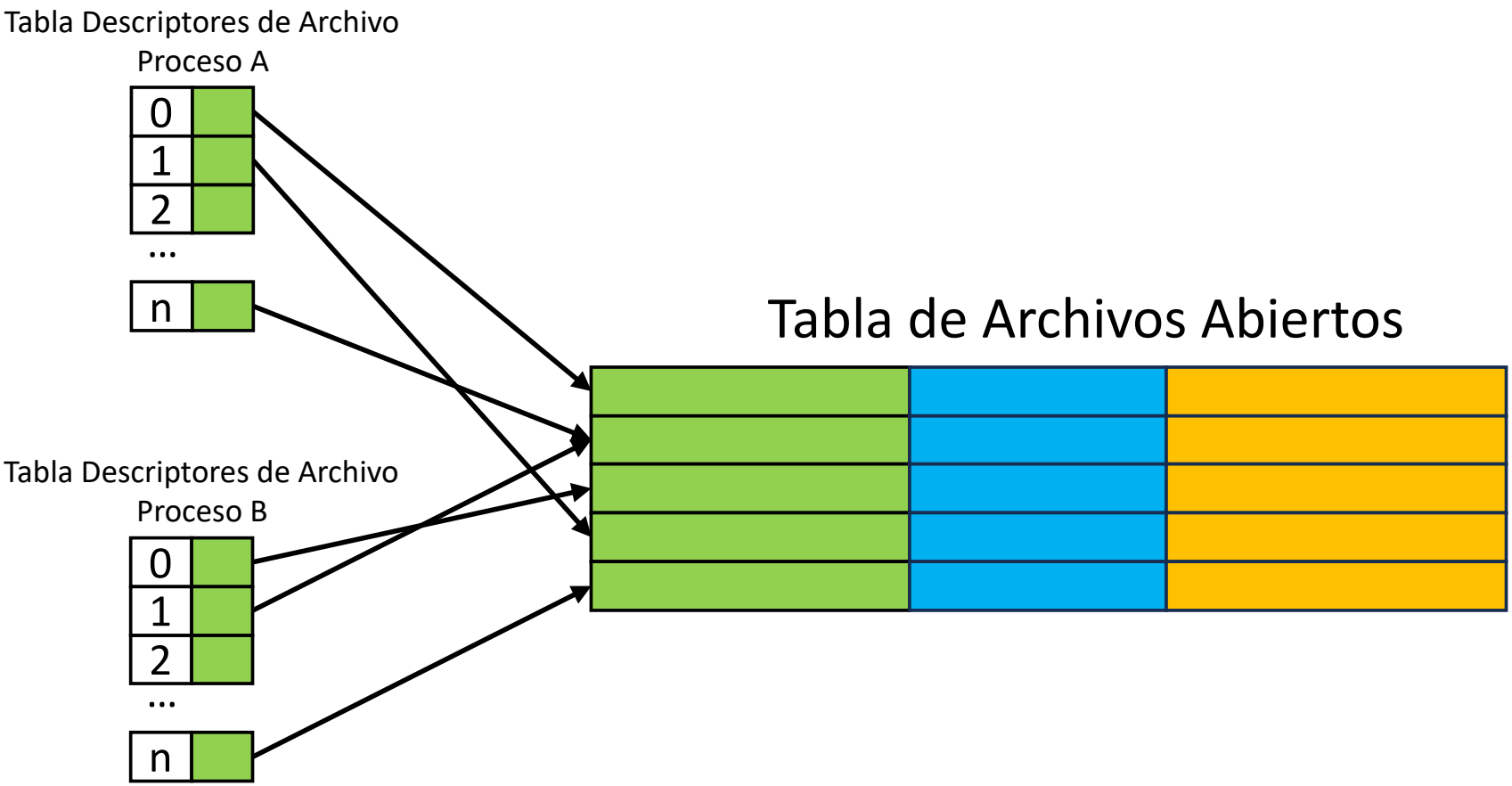
Descriptores de archivo

¿Qué es la Tabla de Descriptores de Archivo?

- La Tabla de Descriptores de Archivos es una tabla única para cada proceso.
- El índice de acceso a esa tabla es del descriptor del archivo (entero que lo identifica) y en cada fila de esa tabla hay un puntero a la Tabla de Archivos Abiertos que permite acceder al archivo para escribir o leer, entre otras operaciones.
- El sistema operativo proporciona una tabla de descriptores de archivo única para cada proceso.

Descriptores de archivo

Tabla de Descriptores de Archivo



Descriptores de archivo

Descriptores de archivo estándar

- Cuando se inicia un proceso, se crean por defecto 3 entradas con `fd` 0 (`stdin`), 1 (`stdout`) y 2 (`stderr`) en la **Tabla de Descriptores de Archivo**.
- Cada una de esas 3 entradas hace referencia a la misma entrada de la **Tabla de Archivos Abiertos**.
- Esa entrada corresponde con un archivo especial llamado `/dev/ttyn`, para `n` $= 1, 2, \ldots$.
- Estos archivos en el listado largo (`ll` o `ls -l`) son de tipo `c` (*character stream*, flujo de caracteres).
- Los archivos `/dev/tty` son archivos especiales que permiten el acceso un dispositivo especial llamado terminal que permite la interacción del usuario mediante el teclado y la pantalla.

Descriptores de archivo

Descriptores de archivo estándar

- Lectura desde `stdin`: consiste en leer desde el descriptor de archivo 0. Cada vez que se escribe un carácter desde el teclado, se lee de `stdin` y se guarda en el archivo `/dev/ttyn`.
- Escritura en `stdout`: consiste en escribir en el descriptor de archivo 1. Cada vez que se escribe en el `stdout` vinculado a un determinado archivo `/dev/ttyn` se escribe en ese terminal.
- Escritura en `stderr`: consiste en escribir en el descriptor de archivo 2. Se usa para escribir errores en el terminal. Cada vez que se escribe en el `stderr` vinculado a un determinado archivo `/dev/ttyn` se escribe en ese terminal.

Descriptores de archivo

Llamadas de entada/salida del sistema

- Cuando se trabaja con archivos en C es más común usar los archivos como flujos (*streams*) de caracteres y usar las funciones `fopen`, `fclose`, `fprintf`, `fscanf`, etc, y el tipo de datos `FILE*`.

- En esta sesión se trata la apertura, escritura y lectura de archivos mediante el uso de descriptores de archivos. Las siguientes funciones requieren la inclusión del fichero de cabecera `unistd.h`

Descriptores de archivo

`creat`: crea un archivo vacío.

```
int creat(char * filename, mode_t mode)
```

Parámetros:

- `filename`: nombre del archivo que se desea crear.
- `mode`: permisos del archivo que se desea crear.

Retorno:

- `int`: descriptor del archivo creado. Un número a partir de 3 (0, 1 y 2 ya están creados). -1 si el archivo no se pudo crear.

Descriptores de archivo

`open`: abre un archivo.

```
int open (const char* Path, int flags [, int mode ]);
```

Parámetros:

- `path`: ruta del archivo.
- `flags`: opciones de apertura. Puede ser la combinación (operador or binario |) de los siguientes flags:
  - `O_RDONLY`: solo lectura.
  - `O_WRONLY`: solo escritura.
  - `O_RDWR`: lectura y escritura.
  - `O_CREAT`: crea el archivo si no existe.
  - `O_EXCL`: previene la creación del archivo si este existe.

Retorno:

- `int`: descriptor del archivo o -1 si hay un error.

Descriptores de archivo

`close`: hace una llamada el sistema para cerrar un archivo dado su descriptor de archivo.

```
int close(int fd);
```

Parámetros:

- `fd`: descriptor del archivo a cerrar.

Retorno:

- `int`: 0 si el cierre se ejecuto con éxito y -1 en caso de error.

Descriptores de archivo

`read`: lee del descriptor de archivo `fd` la cantidad de bytes `cnt` y los guarda en memoria a partir de la dirección de memoria `buf`.

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

Parámetros:

- `fd`: descriptor del archivo.
- `buf`: buffer donde se guarda la información leída.
- `cnt`: longitud del buffer.

Retorno:

- `ssize_t`: devuelve el número de bytes leídos correctamente.
  `ssize_t` es un entero con signo, en el compilador gcc instalado en jair se define como `typedef long ssize_t`
  Devuelve `0` cuando se lea el fin de archivo.
  Devuelve `-1` en caso de error.

Descriptores de archivo

Si se quiere leer una cadena de caracteres previa al pulsado de la tecla `into` de la entrada estándar (`stdin`, descriptor de fichero `0`), `read` no funciona como `scanf` o `gets`, hay que leer carácter a carácter hasta que se encuentre el carácter retorno de carro. Por ejemplo:

```
#define TAM 100 //limite de lectura
int main(){
  char buff[TAM]; // este seria el buffer
  // Lectura por teclado de un caracter
  read(0, &buff[i],1);
  //Fin de lectura cuando se lea el retorno de carro
  while(buff[i] != '\n'){
    read(0,&buff[++i],1);
  }
  // Fin de la cadena
  buff[i]='\0';
}
```

Descriptores de archivo

`write`: escribe `cnt` bytes contenidos en memoria a partir de la dirección `buf` en el archivo cuyo descriptor de archivo sea `fd`.

```
ssize_t write (int fd, void* buf, size_t cnt);
```

Parámetros:

- `fd`: descriptor del archivo.
- `buf`: buffer con la información a guardar.
- `cnt`: número de bytes a escribir.

Retorno:

- `ssize_t`: devuelve el número de bytes escritos correctamente. Devuelve -1 en caso de error.

Procesos pesados

`fork`: crea un nuevo proceso duplicando el proceso actual.

```
pid_t fork(void);
```

Retorno:

- `pid_t`: en el proceso padre, el ID del proceso hijo es retornado. En el proceso hijo, se retorna 0 para indicar que es el proceso hijo. `pid_t` suele ser declarado como un entero.
- -1 en caso de error.

Procesos pesados

- Permite crear procesos hijos.
- Tras su ejecución existen en ejecución dos procesos: padre (el que ejecutó `fork()`) e hijo (el nuevo creado), cuyas imágenes de memoria son básicamente iguales.
- Padre e hijo comparten:
  - Segmento de texto (código).
  - Dispositivos y ficheros que tuviera abiertos el proceso padre.
  - `stdin`, `stdout` y `stderr` (la salida estándar y salida de error estándar de padre e hijo aparecerán mezcladas).

Procesos pesados

- Padre e hijo no comparten:
  - PIDs: el PID de ambos es diferente (son procesos diferentes).
  - *Heap* y pila: la ejecución de `fork()` crea un nuevo espacio en memoria para el *heap* y pila del proceso hijo. En el momento de su creación los valores almacenados en las variables son exactamente iguales que los del padre (se produce un duplicado). Después, cada uno puede ejecutar partes de código diferentes, que afectarán a su zona de memoria (variables), teniendo, entonces, evoluciones y valores diferentes.
- En la llamada a `fork()` el sistema devuelve un 0 al proceso hijo y un valor positivo distinto de 0 (PID del hijo) al padre. Si el valor devuelto es negativo significa que ha habido algún error en la creación del hijo y ésta no se ha podido realizar.

Procesos pesados

`getpid`: obtiene el PID del proceso actual.

```
pid_t getpid();
```

Retorno:

- `pid_t`: el PID del proceso actual.

`getppid`: obtiene el PID del proceso padre del proceso actual.

```
pid_t getppid();
```

Retorno:

- `pid_t`: el PID del proceso padre del proceso actual.

Requieren la inclusión del fichero `#include <unistd.h>`.

Procesos pesados

`wait`: espera a que un proceso hijo termine.

```
pid_t wait(int* status);
```

Si `wait` se encierra en un bucle se hace una espera por cada proceso hijo que termine, si solo hay un proceso hijo y, por ejemplo, dos `wait` seguidos el proceso padre se queda esperando de manera ilimitada.

Parámetros:

- `status`: puntero a una variable donde se almacenará el estado del proceso hijo. Puede ser `NULL` si no se necesita saber el estado.

Retorno:

- `pid_t`: el PID del proceso hijo que terminó.

Requiere la inclusión del fichero `#include <sys/wait.h>`.

Procesos pesados

`fflush`: vacía el búfer asociado a un archivo.

```
int fflush(FILE* stream);
```

Parámetros:

- `stream`: puntero al archivo.

Retorno:

- `int`: 0 si tiene éxito, EOF en caso de error.

Puede usarse con los ficheros especiales `stdin`, `stdout` y `stderr` para vaciar su contenido.

Procesos pesados

`sleep`: hace que el proceso actual pase a estado suspendido (*sleeping*) durante un número especificado de segundos.

```
unsigned int sleep(unsigned int seconds);
```

Parámetros:

- `seconds`: el número de segundos que el proceso debe dormir.

Retorno:

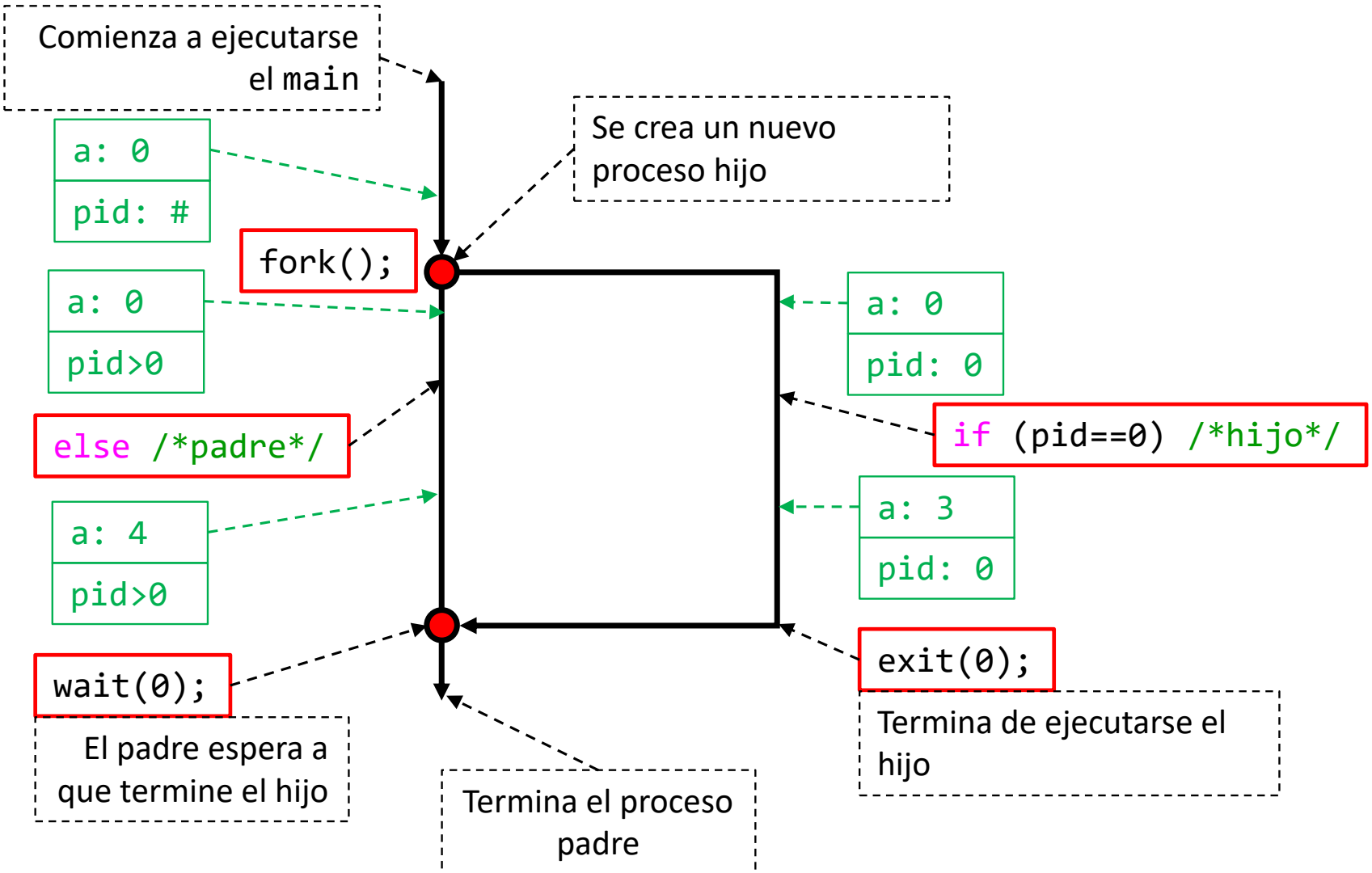
- `unsigned int`: el número de segundos que quedan por dormir, si se interrumpe.

Requiere la inclusión del fichero `#include <unistd.h>`.

Procesos pesados

```c
int main() {
  int pid; inta a=0;
  pid=fork();
  if (pid == -1) {
    printf ("error en creacion de proceso hijo\n");
    exit(1);
  } else {
    if ( pid == 0) /*hijo*/ {
      printf ("Proceso hijo 1\n");
      a = 3;
      fflush(stdout);
      exit(0); /* terminacion con codigo 0 */
    } else /*padre*/ {
      printf ("Proceso padre\n");
      a = 4;
      fflush(stdout);
      wait(0);
    }
  }
}
```

Procesos pesados



Depuración de programas

- En C, cuando un programa imprime en la salida estándar con `printf` o `fprintf`, la salida se almacena en un búfer antes de ser mostrada por el sistema operativo.
- Por este motivo, la salida por consola puede no ser inmediata.
- En ocasiones, por ejemplo, por un error en tiempo de ejecución por un puntero no inicializado, la salida puede perderse, puesto que, aunque el `printf` ya se había ejecutado, no había dado tiempo a imprimirlo en la salida estándar, por lo que el programador puede tener la impresión de que el error era anterior al `printf`, cuando realmente era posterior.
- Para forzar la impresión inmediata, se utiliza `fflush(stdout)`.

Depuración de programas

- Por ejemplo, para un programa con un puntero no inicializado:

```
int *p;
printf("Mensaje por terminal\n");
*p = 5; //esto provoca un error de ejecución y
    puede que no se llegue a visualizar el printf
    en el terminal
```

- Para asegurar la impresión antes de cualquier error, se añade `fflush(stdout)`:

```
int *p;
printf("Mensaje por terminal\n");
fflush(stdout); //la salida del printf saldrá por
    pantalla antes del error en tiempo de
    ejecución
*p = 5;
```

Familia de funciones `exec`

Las funciones `exec` en C son utilizadas para ejecutar un nuevo programa en el contexto del proceso actual.

- `execl`
- `execle`
- `execlp`
- `execv`
- `execve`
- `execvp`

Estas funciones reemplazan la imagen de programa del proceso actual con un nuevo programa especificado por `path` o `file`. Es decir, el programa invocador pierde su código que se reemplaza por el del programa invocado.
La versión con 'l' toma argumentos individuales, mientras que la versión con 'v' toma un array de argumentos. La versión con 'e' permite especificar el entorno de ejecución.

Familia de funciones `exec`

`execl`: reemplaza la imagen del proceso actual con un nuevo programa especificado por la ruta `path` y lo ejecuta.

```
int execl(const char *path, const char *arg0, ..., (
    char *)0);
```

Parámetros:

- `path`: ruta al ejecutable.
- `arg0, ...`: argumentos del nuevo programa.
- `(char *)0`: indicador de fin de lista de argumentos (`NULL`).

Retorno:

- `int`: no retorna en caso de éxito. Si el programa es ejecutado con éxito, la imagen del proceso actual es reemplazada por el nuevo programa y la ejecución continúa a partir de ese punto. Sin embargo, si ocurre algún error durante la ejecución de `execl`, la función retornará -1.

Familia de funciones `exec`

Ejemplo de uso:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
  char *path = "/bin/ls";
  char *comando = "ls";
  char *arg1 ="-l";
  char *arg2 ="/home";
  int estado;

  printf("Antes de llamar a execl()\n");
  if ((estado=execl(path, comando, arg1, arg2, NULL)) == -1)
  {
    printf("El proceso no ha terminado correctamente\n");
    exit(1);
  }
  printf("Esta linea NO se saldra por la salida estandar si
    la llamada a execl() termina correctamente\n");
  return 0;
}
```

Familia de funciones `exec`

Ejemplo de uso con `fork`:

```
int main() {
  char *path = "/bin/ls";
  char *comando = "ls";
  char *arg1 ="-l", *arg2 = "/home";
  int estado;
  printf("\tSe crea otro proceso usando fork()...\n");
  if (fork() == 0) {
    int estado = execl(path, comando, arg1, arg2, NULL);
    printf("\t\tls -l /home  ha tomado el control de este
    proceso hijo. Esto no se ejecutara a menos que termine
    de manera anormal!\n");
    if (estado == -1) {
      printf("\t\tEl proceso no termino correctamente\n");
      exit(1);
    }
  }
  else {
    printf("Esta linea se ejecuta por el proceso padre\n");
    wait(NULL); //espera a que termine el hijo
  }
  return 0;
}
```

Familia de funciones `exec`

`execle`: reemplaza la imagen del proceso actual con un nuevo programa especificado por la ruta `path`. Permite especificar un entorno personalizado.

```
int execle(const char *path, const char *arg0, ..., (
    char *)0, char *const envp[]);
```

Parámetros:

- `path`: ruta al ejecutable.
- `arg0, ...`: argumentos del nuevo programa.
- `(char *)0`: indicador de fin de lista de argumentos.
- `envp[]`: array de *strings* que representan el entorno.

Retorno:

- `int`: sólo retorna en caso de error.

Familia de funciones `exec`

Ejemplo de uso:

```c
#include <unistd.h>

int main(void) {
  char *path = "/bin/sh";
  char *comando = "sh";
  char *arg1 = "-c";
  char *arg2 = "echo $0 $SHELL";
  char *env[] = {"SHELL=Hola", NULL};

  execle(path, comando,arg1, arg2, NULL, env);

  return 0;
}
```

Familia de funciones `exec`

`execvp`: reemplaza la imagen del proceso actual con un nuevo programa especificado por el nombre del archivo y lo ejecuta.

```
int execvp(const char *file, char *const argv[]);
```

Parámetros:

- `file`: nombre del archivo ejecutable.
- `argv`: *array* de argumentos del nuevo proceso.

Retorno:

- `int`: sólo retorna en caso de error.

Familia de funciones `exec`

Ejemplo de uso:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
  char *comando = "ls";
  char *lista_argumentos[] = {"ls", "-l", NULL};
  int estado;

  printf("Antes de llamar a execvp()\n");
  if (( estado  = execvp(comando, lista_argumentos)) == -1)
    {
    printf("El proceso no ha terminado correctamente\n");
    exit(1);
  }
  printf("Esta linea NO se saldra por la salida estandar si
    la llamada a execvp() termina correctamente\n");
  return 0;
}
```

Familia de funciones `exec`

Ejemplo de uso con `fork`:

```
int main() {
  char *comando = "ls";
  char *lista_argumentos[] = {"ls", "-l", NULL};
  int estado;

  printf("\tCreamos otro proceso usando fork()...\n");
  if (fork() == 0) {
    int estado = execvp(comando, lista_argumentos);
    printf("\t\tls -l ha tomado el control de este proceso
    hijo. Esto no se ejecutara a menos que termine de manera
     anormal!\n");
    if (estado == -1) {
      printf("\t\tEl proceso no termino correctamente\n");
      exit(1);
    }
  }
  else {
    printf("Esta linea se ejecuta por el proceso padre\n");
    wait(NULL);
  }
  return 0;
}
```