Bloque I: Principios de sistemas operativos

Notas:

Tema 3. Ficheros

- Conceptos básicos.
- Gestión de ficheros.
- Lectura y escritura.
- Entrada/salida asíncrona.
- Entrada/salida sincronizada.
- Acceso al estado y características de ficheros.
- Funciones de gestión de directorios.
- Tuberías y ficheros especiales FIFO.

1. Conceptos básicos

El sistema de ficheros es una colección de ficheros junto a ciertos atributos que los caracterizan

Proporciona un espacio de nombres, y contiene:

- **ficheros normales: residen en memoria secundaria**
- **directorios**
- **dispositivos orientados al carácter**
- **dispositivos orientados a bloque**
- **tuberías o ficheros especiales FIFO**

Se puede hacer I/O sobre todos ellos, excepto los directorios

Si el sistema de ficheros no existe, el espacio de nombres (sin directorios) y los dispositivos se mantienen

Conceptos básicos (cont.)

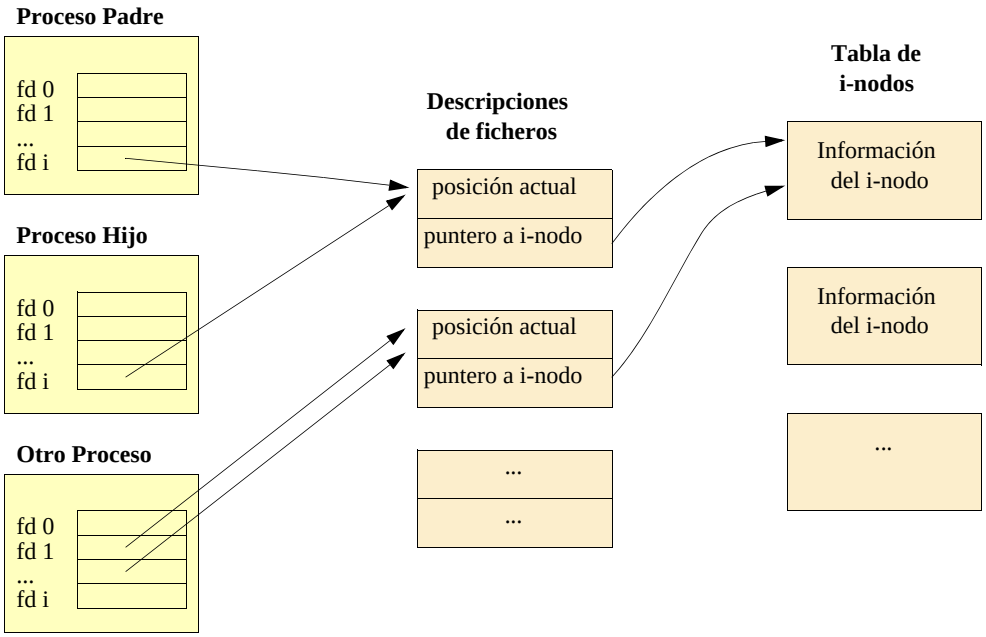
***Descriptor de fichero* (*fd*)**

- **es un entero positivo que identifica un fichero que ha sido abierto para I/O**
- **los descriptores 0, 1, y 2 suelen ser respectivamente la entrada estándar, la salida estándar, y la salida de error**

Descripción de fichero

- **es una estructura de datos perteneciente al núcleo del SO y que existe asociada a un fichero abierto**
- **contiene los punteros de lectura/escritura, etc.**
- **diferentes procesos pueden compartir una misma descripción de fichero**

Compartiendo descripciones de fichero



2. Gestión de ficheros

Abrir un fichero:

```
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *path, int oflag
          [, mode_t mode]);
```

- abre un fichero para leer o escribir
- el `path` puede ser absoluto o relativo al directorio de trabajo
- crea una nueva descripción de fichero y retorna un descriptor de fichero asociado a ella (o `-1` si hay error)

Gestión de ficheros (cont.)

Las opciones definidas para `oflag` permiten:

- abrir para leer, escribir, o ambos
  - `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- añadir al final del fichero
  - `O_APPEND`
- crear el fichero si no existe: requiere parámetro *mode*, que indica los permisos de acceso
  - `O_CREAT`
- truncar (a tamaño 0) el fichero si existe
  - `O_TRUNC`
- I/O bloqueante o no bloqueante, etc.
  - `O_NONBLOCK`

Gestión de ficheros (cont.)

El parámetro `mode` se pone sólo si se crea el fichero, e indica los permisos de lectura (`R`), escritura (`W`), y ejecución (`X`)

- Del propietario (`USRer`)
  - S_IRUSR, S_IWUSR, S_IXUSR
- Del grupo de usuarios (`GRouP`)
  - S_IRGRP, S_IWGRP, S_IXGRP
- Del resto de los usuarios (`OTHers`)
  - S_IROTH, S_IWOTH, S_IXOTH

Gestión de ficheros (cont.)

Crear un fichero:

```
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *path, mode_t mode);
```

- es equivalente a un *open* indicando sólo escritura, creación del fichero, y borrar la información del fichero, si existe
- esta operación sólo se puede hacer si hay sistema de ficheros

Gestión de ficheros (cont.)

Cerrar un fichero:

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

- destruye el descriptor de fichero
- destruye la descripción del fichero si nadie lo tiene abierto

Borrar un fichero

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

- si el fichero está en uso, se borra cuando el último proceso que lo tiene abierto lo cierra

Gestión de ficheros (cont.)

Cambiar de nombre a un fichero

```
#include <stdio.h>
int rename (const char *old, const char *new)
```

- se puede mover el fichero a otro directorio
- se puede cambiar el nombre de un directorio

Modificar el tamaño de un fichero (truncar o expandir)

```
#include <unistd.h>
int ftruncate (int fildes, off_t length)
```

- el nuevo tamaño, en bytes, es `length`

Ejemplo

```
// Crear un fichero con permisos de lectura y escritura para
// el propietario. Poner el tamaño a 1000 bytes y cerrarlo

#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main() {

  int fd;
  off_t longitud=1000;

  fd=creat("pepito",S_IRUSR|S_IWUSR);

  if (fd==-1) {
    perror("Error al crear el fichero");
    exit(1);
  }
```

Ejemplo (cont.)

```
  if (ftruncate(fd,longitud)==-1) {
    perror("Error al cambiar el tamaño");
    exit(1);
  }

  if (close(fd)==-1) {
    perror("Error al cerrar el fichero");
    exit(1);
  }
  exit(0);
}
```


Gestión de ficheros (cont.)

Posicionar el puntero de lectura/escritura

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

- el nuevo puntero dependerá del valor de `whence`:
  - `SEEK_SET`: se pone igual a `offset`
  - `SEEK_CUR`: se le añade `offset`
  - `SEEK_END`: se pone igual al tamaño del fichero + `offset`
- devuelve el puntero resultante (número de bytes desde el principio del fichero)

Gestión de ficheros (cont.)

Crear un enlace directo (*link*) a un fichero

```
#include <unistd.h>
int link (const char *existing, const char *new)
```

- es una forma de dar dos nombres a un mismo fichero
- se puede hacer simbólico (el sistema almacena el nombre del fichero destino, no una referencia a él)

```
#include <unistd.h>
int symlink(const char *path1, const char *path2);
```

Duplicar un descriptor de fichero

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

- retorna un descriptor referido al mismo fichero que `fildes`

3. Lectura y escritura

Leer:

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

- Intenta leer `nbyte` bytes de fichero especificado almacenándolos en el buffer apuntado por `buf`
- retorna el número de bytes leidos (<=`nbyte`)
- la lectura puede ser bloqueante o no, según lo definido en el `open`

Lectura y escritura (cont.)

Escribir:

```
#include <unistd.h>
ssize_t write (int fildes, const void *buf,
               size_t nbyte);
```

- Intenta escribir **nbyte** bytes del buffer apuntado por ***buf*** en el fichero especificado por **fildes**
- retorna el número de bytes escritos (<**nbyte** si no caben)
- la escritura puede ser bloqueante o no, según lo definido en el **open**

Ejemplo de lectura/escritura: fichero "alumnos.h"

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

typedef struct {
  char nombre[30];
  char dni[10];
} alumno;

#define OK                0
#define ERROR_DE_LECTURA  1
#define ERROR_DE_ESCRITURA 2

void teclea_alumno(alumno *a);

void muestra_alumno(const alumno *a);

int escribe_alumno(const alumno *a, off_t pos, int fd);

int lee_alumno(alumno *a, off_t pos, int fd);
```

Ejemplo de lectura/escritura: fichero "alumnos.c"

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

void teclea_alumno(alumno *a) {
  char tonto [80];

  printf("Nombre: ");
  fgets(a->nombre,30,stdin);
  printf("DNI: ");
  scanf("%s",a->dni);
  // quitar el salto de linea
  fgets(tonto,80,stdin);
}
```


Ejemplo de lectura/escritura: fichero "alumnos.c" (cont.)

```
void muestra_alumno(const alumno *a) {
  printf("Nombre: %s\n",a->nombre);
  printf("DNI: %s\n\n",a->dni);
}

int escribe_alumno(const alumno *a, off_t pos, int fd) {
  ssize_t escritos;

  if (lseek(fd,pos,SEEK_SET)==-1) {
    return ERROR_DE_ESCRITURA;
  }
  escritos=write(fd,a,sizeof(alumno));
  if (escritos<sizeof(alumno)) {
    return ERROR_DE_ESCRITURA;
  } else{
    return OK;
  }
}
```

Ejemplo de lectura/escritura: fichero "alumnos.c" (cont.)

```
int lee_alumno(alumno *a, off_t pos, int fd) {
  ssize_t leidos;

  if (lseek(fd,pos,SEEK_SET)==-1) {
    return ERROR_DE_LECTURA;
  }
  leidos=read(fd,a,sizeof(alumno));
  if (leidos<sizeof(alumno)) {
    return ERROR_DE_LECTURA;
  } else {
    return OK;
  }
}
```


Ejemplo de lectura/escritura: fichero "prueba_alumnos.c"

```
#include "alumnos.h"
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define SALIR 0
#define NUEVO 1
#define MIRAR 2
```

Ejemplo de lectura/escritura: fichero "prueba_alumnos.c"

```
int menu() {
  char tonto [80];
  int opcion=-1; // indica error si la lectura falla

  printf("0 - Salir\n");
  printf("1 - Nuevo Alumno\n");
  printf("2 - Mirar un Alumno\n");
  printf("Introduce opcion: ");
  scanf("%d",&opcion);
  // quitar el salto de linea
  fgets(tonto,80,stdin);
  return opcion;
}
```


Ejemplo de lectura/escritura: fichero "prueba_alumnos.c"

```
off_t lee_pos() {
  int pos=-1; // si hay error al leer devuelve -1
  char tonto [80];

  printf("Introduce posicion: ");
  scanf("%d",&pos);
  // quitar el salto de linea
  fgets(tonto,80,stdin);
  return (off_t) pos;
}

int main() {
  int opcion;
  int fd;
  off_t pos;
  alumno al;

  fd=open("alumnos",O_RDWR);
```

Ejemplo de lectura/escritura: fichero "prueba_alumnos.c"

```
  if (fd==-1) {
    perror("Error al abrir");
    exit(1);
  }
  do {
    opcion=menu();
    switch (opcion) {
    case SALIR:
      break;
    case NUEVO:
      teclea_alumno(&al);
      pos=lee_pos();
      if ((pos<0) || escribe_alumno(&al,pos,fd)!=OK) {
         printf("posicion incorrecta o error de escritura\n");
      }
      break;
```


Ejemplo de lectura/escritura: fichero "prueba_alumnos.c"

```
    case MIRAR:
      pos=lee_pos();
      if ((pos<0) || lee_alumno(&al,pos,fd)!=OK) {
        printf("posicion incorrecta o error de lectura\n");
      } else {
        muestra_alumno(&al);
      }
      break;
    default:
      printf("Opcion erronea\n\n");
    }
  } while (opcion!=0);
  close(fd);
  exit(0);
}
```

4. I/O asíncrona con threads

Es posible hacer entrada/salida asíncrona haciendo la operación de lectura o escritura desde un thread creado al efecto

El siguiente ejemplo lee datos del teclado asíncronamente

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

// estructura de datos para la operación de lectura
typedef struct {
  int fildes;
  void *buf;
  int nbytes;
  ssize_t numread;
} async_data;
```


I/O asíncrona con threads (cont.)

```
// thread de lectura
void * async_reader (void * arg) {

  ((async_data *) arg)->numread=
    read(((async_data *)arg)->fildes,((async_data *) arg)->buf,
        ((async_data *) arg)->nbytes);
  pthread_exit(NULL);
}

// programa principal
int main ()
{
  char str_read [10];
  pthread_t th;
  async_data my_aiocb;
```

I/O asíncrona con threads (cont.)

```
  // Preparar los datos para I/O asíncrona
  my_aiocb.fildes=0;
  my_aiocb.buf=&str_read;
  my_aiocb.nbytes=8;
  my_aiocb.numread=0;

  // Comenzar la lectura asíncrona
  pthread_create(&th,NULL,async_reader,&my_aiocb);
    // Iterar hasta que la operación se complete
  while (my_aiocb.numread==0) {
    // Este código se ejecuta en concurrencia con la lectura
    printf("Operation in progress\n");
    sleep(1);
  }
  printf("Number of bytes read=%d; string=",my_aiocb.numread);
  str_read[my_aiocb.numread]=0;
  printf("%s\n",str_read);
  exit(0);
}
```


5. Entrada/salida sincronizada

Permite garantizar que los datos han sido transferidos con éxito al o del dispositivo físico

"*Transferidos con éxito*": se pueden leer después de hacer un *open*, incluso después de un fallo del sistema o de potencia

Existen dos niveles de sincronización:

- *de datos*: los datos se transfieren con éxito, así como la información de directorio necesaria para acceder a ellos
- *de fichero*: además de lo anterior, se transfieren con éxito los atributos del fichero (hora de modificación, etc.)

La lectura sincronizada implica además que se han realizado previamente todas las operaciones de escritura pendientes

Interfaz para I/O sincronizada

Al abrir un fichero, se pueden especificar las siguientes opciones:

- **O_DSYNC**: las operaciones de escritura se harán con sincronización de datos
- **O_SYNC**: las operaciones de escritura se harán con sincronización de fichero
- **O_RSYNC**: las operaciones de lectura se harán con sincronización de datos o fichero, según la presencia de las opciones **O_DSYNC** u **O_SYNC**

Interfaz para I/O sincronizada (cont.)

Se puede pedir explícitamente la sincronización:

```
#include <unistd.h>
int fsync (int fildes); // de fichero
int fdatasync (int fildes); // de datos
```

También se pueden sincronizar operaciones de I/O asíncrona:

```
#include <aio.h>
int aio_fsync (int op, struct aiocb *aiocbp)
```

- **op** puede ser **O_DSYNC** u **O_SYNC**

6. Acceso al estado y características de ficheros

Algunos campos de `struct stat`:

Tipo	Nombre	Descripción
`mode_t`	`st_mode`	Modo, incluidos los permisos de uso
`uid_t`	`st_uid`	Identificador del usuario o propietario
`gid_t`	`st_gid`	Identificador del grupo
`off_t`	`st_size`	Tamaño, en bytes
`time_t`	`st_atime`	Fecha y hora del último acceso
`time_t`	`st_mtime`	Fecha y hora de la última modificación
`time_t`	`st_ctime`	Fecha y h. de creación o cambio de estado

El tipo `time_t` es un entero que representa segundos desde la época (00:00h del 1 de enero de 1970)

Estado y características de ficheros (cont.)

A partir del modo se pueden obtener:

- permisos de uso (con las constantes vistas en `open()`)
- tipo de fichero: con las siguientes macros de test, siendo `m` el modo:

Macro	Descripción
`S_ISDIR(m)`	Es un directorio
`S_ISCHR(m)`	Es un dispositivo de caracteres
`S_ISBLK(m)`	Es un dispositivo de bloques
`S_ISREG(m)`	Es un fichero normal
`S_ISFIFO(m)`	Es una tubería o fichero FIFO

Estado y características de ficheros (cont.)

Leer el estado de un fichero

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *path, struct stat *buf)
int fstat (int fildes, struct stat *buf)
```

Modificar el modo de un fichero

```
#include <sys/stat.h>
int chmod (const char *path, mode_t *mode)
int fchmod (int fildes, mode_t *mode)
```

Cambiar el propietario y el grupo

```
#include <sys/types.h>
int chown (const char *path, uid_t owner, gid_t group)
```


7. Funciones de gestión de directorios

Abrir un directorio

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *dirname)
```

- Abre el directorio para poder recorrerlo desde el principio

Recorrer un directorio

```
int readdir_r (DIR *dirp, struct dirent *entry,
               struct dirent **result)
```

- lee una entrada del directorio en `entry` y coloca un puntero a esa entrada en `result` (o `NULL` si no hay más)

```
void rewinddir (DIR *dirp)
```

- prepara `dirp` para recorrerlo otra vez desde el principio

Funciones de gestión de directorios (cont.)

Estructura `dirent`:

```
struct dirent {
  char[] d_name;
}
```

- El string puede tener hasta `NAME_MAX+1` caracteres

Cerrar un directorio

```
int closedir (DIR *dirp)
```

Cambiar el directorio de trabajo

```
int chdir (const char *path)
```

Obtener el directorio de trabajo

```
char *getcwd (char *buf, size_t size);
```


Ejemplo: recorrer el directorio de trabajo

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

int main() {
  char wd[PATH_MAX]; // PATH_MAX es el pathname máximo
  struct dirent entry;
  struct dirent *result;
  char * wd_ptr;
  DIR * dir_ptr;

  // leer directorio de trabajo
  wd_ptr=getcwd(wd,PATH_MAX);
  if (wd_ptr==NULL) {
    perror("Error al leer directorio de trabajo\n");
    exit(1);
  }
```

Ejemplo (cont.)

```
  // abrir el directorio
  dir_ptr=opendir(wd);
  if (dir_ptr==NULL){
    perror("Error al abrir el directorio\n");
    exit(1);
  }
  // recorrer el directorio
  do {
    if (readdir_r(dir_ptr,&entry,&result)!=0) {
      printf("Error al leer el directorio\n");
      exit(1);
    }
    if (result!=NULL) {
      printf("Nombre fichero: %s\n",entry.d_name);
    }
  } while (result!=NULL);
  closedir(dir_ptr);
  exit(0);
}
```


Funciones de gestión de directorios (cont.)

Crear un directorio

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode)
```

Borrar un directorio

```
int rmdir (const char *path)
```

8. Tuberías y ficheros especiales FIFO

Permiten el intercambio de datos entre procesos, mediante una cola donde se leen y escriben bytes

Crear un fichero especial FIFO (es como una tubería, pero con nombre)

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *path, mode_t mode)
```

Crear una tubería (“pipa”)

```
int pipe (int fildes[2])
```

- devuelve en `fildes` dos descriptores de fichero, para los extremos de lectura y escritura

Ejemplo con tuberías

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

int main() {

  int i;
  char mensaje[30];
  int tuberia[2];
  pid_t child;

  // crea la tubería y luego el proceso hijo la hereda
  // tratamiento de errores omitido
  pipe(tuberia);
  child=fork();
```

Ejemplo (cont.)

```
  if (child==0) {
    //proceso hijo lee de la tubería 30 mensajes
    for (i=0; i<30; i++) {
      read(tuberia[0],mensaje,30);
      printf("Mensaje %d = %s\n",i,mensaje);
    }
    exit(0);
  } else {
    //proceso padre escribe en la tubería 30 mensajes
    for (i=0; i<30; i++) {
      strncpy(mensaje,"Esto es un mensaje",30);
      write(tuberia[1],mensaje,30);
    }
    exit(0);
  }
}
```