Master en Computación

Plataformas de Tiempo Real

POSIX Avanzado y Extensiones

Tema 1. Ficheros y entrada/salida

Tema 2. Gestión de Interrupciones en MaRTE OS
Tema 3. Monitorización y control del tiempo de ejecución
Tema 4. Planificación EDF
Tema 5. Planificación a Nivel de Aplicación

Tema 1. Ficheros y entrada/salida

1.1. Conceptos básicos
1.2. Gestión de ficheros
1.3. Llamadas para entrada/salida
1.4. I/O asíncrona con threads
1.5. Entrada/salida sincronizada
1.6. Funciones de gestión de directorios
1.7. Tuberías y ficheros especiales FIFO

1.1 Conceptos básicos

***Fichero*: objeto del sistema operativo que se puede leer o escribir, y que puede significar varias cosas:**

- **fichero normal: representa información (programas o datos) almacenada en el disco**
- **directorio**
- **fichero especial de dispositivo orientado a caracteres**
  Único tipo de fichero obligatorio en el perfil mínimo
- **fichero especial de dispositivo orientado a bloques**
- **tubería o fichero especial FIFO**
- **opcionalmente: colas de mensajes, semáforos, objetos de memoria compartida**

• **cada fichero se identifica mediante un nombre (*pathname*)**

Sistema de ficheros

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 (*sistema mínimo*)**
- **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 (*sistema mínimo*), el espacio de nombres (sin directorios) y los dispositivos se mantienen

Descriptor de fichero (`fd`)

- **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

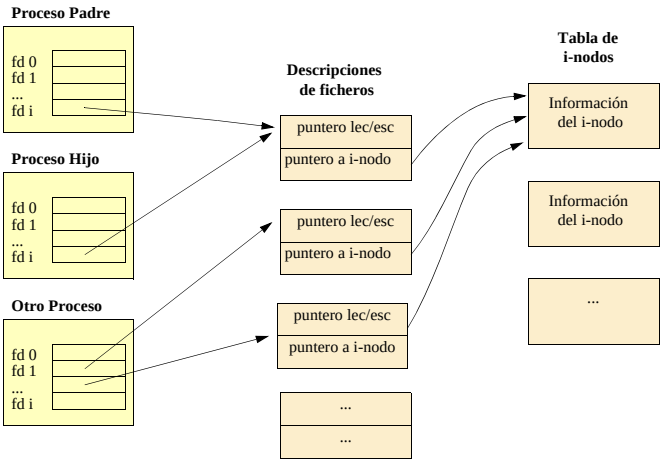
- **Estructura de datos perteneciente al *kernel* y que existe asociada a un fichero abierto**
- **contiene: punteros de lectura/escritura, modo de acceso y puntero al *i-node* (*MaRTE OS: sólo el modo de acceso*)**

i-node

- **Contiene la información relativa al fichero: localización en el disco, tamaño, propietario, grupo, permisos, número de *links*, etc.**

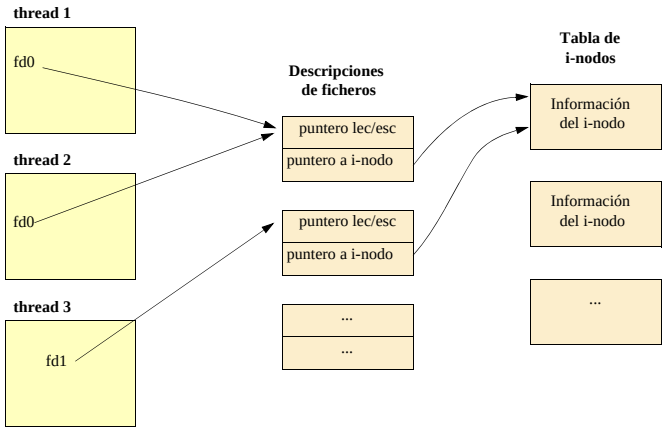
Compartiendo descripciones de fichero

Entre procesos:



Compartiendo descripciones de fichero

Entre threads del mismo proceso:



1.2 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**
- **en caso de error retorna `-1` y pone la variable `errno` al valor apropiado**
  - **algunos errores: `EACCES`, `EEXIST`, ... (ver la página del manual para un listado completo)**

Gestión de ficheros (cont.)

Las opciones definidas para `oflag` permiten:

- **abrir para leer, escribir, o ambos (*sistema mínimo*)**
  - **`O_RDONLY`, `O_WROLNY`, `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. (*sistema mínimo*)**
  - **`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`**

Este parámetro no se utiliza en el perfil de "Sistema de Tiempo Real Mínimo"

En los sistemas mínimos no se pueden crear 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**

Crear **un fichero: (*NO en sistemas mínimos*)**

```
#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**
- **sólo se puede hacer si hay sistema de ficheros**

Gestión de ficheros (cont.)

Borrar **un fichero: (*NO en sistemas mínimos*)**

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

- **se borra cuando lo cierra el último proceso**

Cambiar de nombre **a un fichero: (*NO en sistemas mínimos*)**

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

- **para ficheros y directorios**

Modificar el tamaño **de un fichero: (*NO en sistemas mínimos*)**

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

1.3 Llamadas para entrada/salida

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 leídos (<=`nbyte`)
- retorna 0 cuando se ha llegado al fin de un fichero
- bloqueante o no, según lo definido en el open:
  - Bloqueante (sin `O_NONBLOCK`): el thread se bloquea hasta que se pueda leer *algún* dato
  - No Bloqueante (con `O_NONBLOCK`): lee los bytes que puede. Si no se puede leer ninguno retorna -1 y pone `errno` al valor `EAGAIN`

Llamadas para entrada/salida (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)
- bloqueante o no, según lo definido en el open:
  - Bloqueante (sin `O_NONBLOCK`): el thread se bloquea hasta que se puedan escribir *todos* los datos
  - No Bloqueante (con `O_NONBLOCK`): escribe los bytes que puede. Si no se puede escribir ninguno retorna -1 y pone `errno` al valor `EAGAIN`

Ejemplo: Llamadas para entrada/salida

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

int main () {
  int fd; // descriptor de fichero
  char *str = "hola";  char str_leido[10];
  int num = 4;         int num_leido;
  // crea el fichero para escritura
  CHKE( fd=open("pru.dat", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR) );
  // escribe el string
  if (write (fd, str, 5) != 5)
    printf ("Error escribiendo string en el fichero \n");
  // escribe el número
  if (write (fd, &num, sizeof(int)) != sizeof(int))
    printf ("Error escribiendo numero en el fichero \n");
  // cierra el fichero
  CHKE( close(fd) );
```

```
   // Lectura de los datos

   // abre el fichero para lectura
   CHKE( fd = open("pru.dat", O_RDONLY) );


   if (read (fd, str_leido, 5) != 5)
     printf ("Error leyendo string del fichero \n");

   if (read (fd, &num_leido, sizeof(int)) != sizeof(int))
     printf ("Error leyendo numero del fichero \n");

   CHKE( close(fd) );

   printf ("Leído string:%s y num:%d\n", str_leido, num_leido);

   return 0;
}
```


1.4 I/O asíncrona con threads

El estándar POSIX proporciona funciones para realizar operaciones de entrada/salida asíncrona (AIO)

- **Cada operación de AIO opera en paralelo con la aplicación**
  - **envía una señal para informar que ha terminado**
- **Servicio es opcional (*NO en sistemas mínimos*)**

Funciones entrada/salida asícrona:

```
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);

struct aiocb {
   int             aio_fildes;     /* File descriptor */
   off_t           aio_offset;     /* File offset */
   volatile void  *aio_buf;        /* Location of buffer */
   size_t          aio_nbytes;     /* Length of transfer */
   int             aio_reqprio;    /* Request priority */
   struct sigevent aio_sigevent;   /* Notification method */
   int             aio_lio_opcode; /* Operation to be performed;
                                      lio_listio() only */
};
```


I/O asíncrona con threads:

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

- **sólo el thread que hace una operación de E/S se bloquea, los demás pueden seguir ejecutando**

El siguiente ejemplo lee datos del teclado asíncronamente utilizando un thread

Ejemplo: I/O asíncrona con threads:

```
#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_t;


// thread de lectura
void * async_reader (void *arg) {
  async_data_t *async_data = (async_data_t *) arg;

  async_data->numread=
    read(async_data->fildes, async_data->buf, async_data->nbytes);

  return NULL;
}
```



```
int main ()
{
  char str_read [10];
  pthread_t th;
  async_data_t my_async_data;

  // Prepara los datos para I/O asíncrona
  my_async_data.fildes=0;  my_async_data.buf=&str_read;
  my_async_data.nbytes=8;  my_async_data.numread=0;

  CHK( pthread_create(&th, NULL, async_reader, &my_async_data) );

  while (my_async_data.numread==0) {
    // mientras operación en marcha este código se ejecuta en
    // concurrencia con la lectura
    printf("Operation in progress\n");
    sleep(1);
  }

  printf("Number of bytes read=%d; string=", my_async_data.numread);
  str_read[my_async_data.numread]=0;
  printf("%s\n", str_read);

  return 0;
}
```


1.5 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

En MaRTE la I/O es siempre sincronizada

Al abrir un fichero, se pueden especificar las 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`**
  - **la lectura es completa cuando los datos han sido transferidos a la memoria del proceso**
  - **si había alguna operación de escritura pendiente, esta se finaliza antes de la lectura**

Se puede pedir explícitamente la sincronización:

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

- **sincroniza a nivel de fichero todas las operaciones I/O pendientes**

```
int fdatasync (int fildes);
```

- **sincroniza a nivel de datos todas las operaciones I/O pendientes**

1.6 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)
```

- **copia la siguiente entrada del directorio en `entry` y coloca un puntero a esa entrada en `result` (o `NULL` si no hay más)**

Prepara `dirp` para recorrerlo otra vez desde el principio

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

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>
#include "error_checks.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);
  }
```



```
  // abrir el directorio
  dir_ptr=opendir(wd);
  if (dir_ptr==NULL){
    perror("Error al abrir el directorio\n");
    exit(1);
  }

  // recorrer el directorio
  do {
    CHKE (readdir(dir_ptr, &entry, &result) );

    if (result!=NULL) {
      printf("Nombre fichero: %s\n",entry.d_name);
    }
  } while (result!=NULL);

  // cierra el directorio
  closedir(dir_ptr);
  exit(0);
}
```

Crear y borrar un directorio

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)
```


1.7 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>
#include "error_checks.h"

int main() {

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

  // crea la tubería y luego el proceso hijo la hereda
  CHKE( pipe(tuberia) );
  child = fork();
```

```
    if (child==0) {

      //proceso hijo lee de la tubería 20 mensajes
      for (i=0; i<20; i++) {
        read(tuberia[0], mensaje, 30);
        printf("Mensaje %d = %s\n", i, mensaje);
      }
      exit(0);

    } else {

      //proceso padre escribe en la tubería 20 mensajes
      for (i=0; i<20; i++) {
        strncpy(mensaje,"Esto es un mensaje", 30);
        write(tuberia[1], mensaje, 30);
      }
      exit(0);
    }
  }
```