GESTION DE FICHEROS (II)

Guía para la práctica:

Implementación de un sistema de gestión de ficheros basado en UNIX

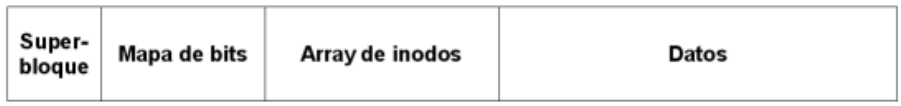
Sistema de ficheros basado UNIX

- Los sistemas de ficheros suelen estar situados en dispositivos de almacenamiento modo bloque, tales como discos o cintas.
- Un sistema de ficheros se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene un tamaño fijo (homogéneo).
  - El tamaño del bloque es el mismo para todo el sistema de ficheros y suele ser múltiplo de 512.
- Unix contempla todos los ficheros como flujos de bytes.
- Existe una estructura llamada i-nodo que contiene toda la información referente a un fichero.

Sistema de ficheros de UNIX

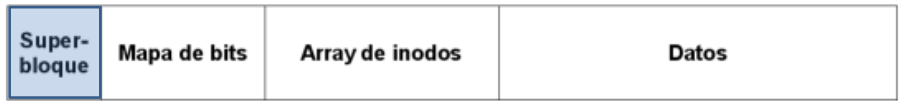
- El tamaño elegido para el bloque va a influir en las prestaciones globales del sistema.
  - Bloques grandes ⇒ velocidad de transferencia entre el disco y la memoria grande.
  - Si demasiado grandes ⇒ capacidad de almacenamiento del disco desaprovechada cuando abundan los archivos pequeños que no llegan a ocupar un bloque completo.
  - Valores típicos para el tamaño del bloque: 512B, 1KB, 2KB, 4KB

Organización de un dispositivo



- El **superbloque** es un bloque que contiene información general sobre el sistema de ficheros.
- En el **mapa de bits** hay un bit por cada bloque del sistema de ficheros; valdrá 0 para bloques libres y 1 para bloques ocupados.
- Un **inodo** contiene las características de un directorio o fichero del sistema de ficheros.
  - La cantidad de inodos por nosotros definida indica la máxima cantidad de directorios o ficheros que pueden llegar a existir.
  - Los inodos se guardan en el **array de inodos**.
    Dentro del array de inodos, los inodos libres se organizan como una lista enlazada (mientras que los inodos ocupados apuntan a directorios o ficheros existentes).
- Los directorios y ficheros en sí se almacenan en la zona de **datos**.

El superbloque



- Es un bloque que contiene información general sobre el sistema de ficheros.
  - **Posición del primer bloque del mapa de bits**
    - SB.posPrimerBloqueMB = posSB + tamSB //posSB = 0, tamSB = 1
  - **Posición del último bloque del mapa de bits**
    - SB.posUltimoBloqueMB = SB.posPrimerBloqueMB + tamMB - 1
  - **Posición del primer bloque del array de inodos**
    - SB.posPrimerBloqueAI = SB.posUltimoBloqueMB + 1
  - **Posición del último bloque del array de inodos**
    - SB.posUltimoBloqueAI = SB.posPrimerBloqueAI + tamAI - 1

El superbloque

- **Posición del primer bloque de datos**
  - SB.posPrimerBloqueDatos = SB.posUltimoBloqueAI + 1
- **Posición del último bloque de datos**
  - SB.posUltimoBloqueDatos = nbloques - 1
- **Posición del inodo del directorio raíz**
  - SB.posInodoRaiz = 0
- **Posición del primer inodo libre**
  - Inicialmente SB.posPrimerInodoLibre = 0.
  - Tras crear el Directorio raíz pasará a valer 1.
  - Posteriormente se irá actualizando para apuntar a la cabeza de la lista de inodos libres (mediante las llamadas a reservar_inodo() y liberar_inodo())

El superbloque

- **Cantidad de bloques libres**
  - Inicialmente: SB.cantBloquesLibres = nbloques
  - Cuando indiquemos en el mapa de bits los bloques que ocupan el SB, el propio MB y el AI, restaremos esos bloques de la cantidad de bloques libres
  - Cuando reservemos un bloque ⇒ SB.cantBloquesLibres--
  - Cuando liberemos un bloque ⇒ SB.cantBloquesLibres++
- **Cantidad de inodos libres**
  - Inicialmente: SB.cantInodosLibres = ninodos (el 1er inodo será para el directorio raíz)
  - Cuando reservemos un inodo ⇒ SB.cantInodosLibres--
  - Cuando liberemos un inodo ⇒ SB.cantInodosLibres++
- **Cantidad total de bloques**
  - Se pasará como argumento de línea de comandos al inicializar el sistema ⇒ $ ./mi_mkfs <nombre_fichero> <cantidad_bloques>
- **Cantidad total de inodos**
  - Lo determina el administrador del sistema Ej:ninodos = nbloques/4

El superbloque

- Al arrancar el sistema se montan los sistemas de archivos locales y remotos y se lee el superbloque a memoria
- Cada vez que desde un proceso se accede a un archivo, es necesario consultar el superbloque y la lista de inodos (normalmente el kernel realizará la E/S con el disco a través del buffer caché)

Mapa de bits

- En el mapa de bits hay un bit por cada bloque del sistema de ficheros; valdrá 0 para bloques libres y 1 para bloques ocupados.
- Tamaño en bloques del mapa de bits:

```
si ((nbloques/8) % BLOCKSIZE) = 0
  tamMB = (nbloques/8) / BLOCKSIZE
si no
  tamMB = ((nbloques/8) / BLOCKSIZE) + 1
```

Ejemplos:

nbloques = 500.000, BLOCKSIZE = 1024 bytes ⇒ tamMB = 62 bloques

nbloques = 10.000, BLOCKSIZE = 1024 bytes ⇒ tamMB = 2 bloques

Mapa de bits. Operaciones

- **Ubicación del bit a partir del nº de bloque**
  - Posición del byte en el mapa de bits:

    posbyte = nbloque / 8
  - Posición del bit dentro del byte:

    posbit = nbloque % 8

  Ejemplos:
  - nbloque = 6 ⇒ posbyte = 0 y posbit = 6
  - nbloque = 19 ⇒ posbyte = 2 y posbit = 3
  - nbloque = 16.000⇒ posbyte = 2.000 y posbit = 0
  - Bloque del mapa de bits donde se halla ese bit:

    bloqueMB = posbyte/BLOCKSIZE
  - Bloque del dispositivo donde leer/escribir el bit:

    bloque = bloqueMB + SB.posPrimerBloqueMB

Mapa de bits. Operaciones

- **Poner a 1 el bit correspondiente a la posición de un bloque determinado**

```
unsigned char mascara = 128;            // 10000000
mascara >>= posbit;    // desplazamiento de bits a la derecha
bufferMB[posbyte] |= mascara;      // operador OR para bits
```

Ejemplo: posbit=3

- mascara: 10000000
- Desplazando el 1er bit de la máscara a la derecha 3 posiciones ⇒ mascara: 00010000
- Si hacemos el OR binario de la máscara con el byte del mapa de bits, obtendremos un 1 en la posición=3 y preservaremos el valor del resto:

00010000 | xxx0xxxx = xxx1xxxx
00010000 | xxx1xxxx = xxx1xxxx

Observación: Como bufferMB es un array del tamaño de un bloque, 1º hay que relativizar la posición de posbyte:

posbyte = posbyte % BLOCKSIZE

Mapa de bits. Operaciones

- **Poner a 0 el bit correspondiente a la posición de un bloque determinado**

```
unsigned char mascara = 128;            // 10000000
mascara >>= posbit;    // desplazamiento de bits a la derecha
bufferMB[posbyte] &= ~mascara; // AND y NOT para bits
```

Ejemplo: posbit=3

- mascara: 10000000
- Desplazando el 1er bit de la máscara a la derecha 3 posiciones ⇒ mascara: 00010000
- Si hacemos el NOT binario de la máscara ⇒ mascara: 11101111
- Si hacemos el AND binario de la máscara con el byte del mapa de bits, obtendremos un 0 en la posición=3 y preservaremos el valor del resto:

11101111 & xxx0xxxx = xxx0xxxx
11101111 & xxx1xxxx = xxx0xxxx

Mapa de bits. Operaciones

- **Leer el bit correspondiente a la posición de un bloque determinado**

```
unsigned char mascara= 128;  // 10000000
mascara >>= posbit;              // desplazamiento de bits a la derecha
mascara &= bufferMB[posbyte];       // operador AND para bits
mascara >>= (7 - posbit);          // desplazamiento de bits a la derecha
```

Ejemplo: posbit=3

- mascara: 10000000
- Desplazando el 1er bit de la máscara a la derecha 3 ⇒ mascara: 00010000
- Si hacemos el AND binario de la máscara con el byte del mapa de bits, obtenemos el bit de la posición=3 y el resto queda a 0:

  00010000 & xxx0xxxx = 00000000
  00010000 & xxx1xxxx = 00010000
- En el byte resultado obtenido hacemos un desplazamiento de 7-posbit posiciones a la derecha, o sea de 4 y así nos queda:

  00000000 si originariamente había un 0 en el mapa de bits // 0 en decimal
  00000001 si originariamente había un 1 en el mapa de bits // 1 en decimal

Mapa de bits. Operaciones

- **Encontrar el primer bit a 0 (por la izquierda) de un byte**

```
unsigned char mascara = 128;   // 10000000
int i = 0;
if (byte < 255) {                  // hay bits a 0 en el byte
while (byte & mascara) {           // operador AND para bits
      i++;
       byte <<= 1;                 // desplazamiento de bits a la izquierda
}
return i;
```

Ejemplo: byte= 251 // 11111011 en binario

- mascara: 10000000, i = 0
- Vamos haciendo en AND binario del byte del mapa de bits con la máscara, incrementamos el contador y desplazamos un bit a la izquierda:

  11111011 & 10000000 = 10000000, i = 1
  11110110 & 10000000 = 10000000, i = 2
  11101100 & 10000000 = 10000000, i = 3
  11011000 & 10000000 = 10000000, i = 4
  10110000 & 10000000 = 10000000, i = 5
  01100000 & 10000000 = 00000000 //fin
- El primer 0 está en la posición 5 (contando desde la izqda y empezando por 0)

Array de inodos

- Cada archivo en un sistema UNIX tiene asociado un inodo.
- El array de inodos tiene una entrada por cada archivo, donde se guarda una descripción del mismo.
- El administrador del sistema es el encargado de especificar el tamaño de la lista de inodos al configurar el sistema (máxima cantidad de directorios o ficheros que pueden llegar a existir).
  - Para nuestra práctica podemos usar por ejemplo:

    ninodos = nbloques/4

Array de inodos

- Tamaño, en bloques, del array de inodos:

```
si ((ninodos * TAMINODO) % BLOCKSIZE) = 0
  tamAI = (ninodos * TAMINODO) / BLOCKSIZE
si no
  tamMB = (ninodos * TAMINODO) / BLOCKSIZE) + 1
```

Ejemplos:

ninodos= 5.000, TAMINODO = 128 bytes, BLOCKSIZE = 1.024 ⇒ tamAI = 625 bloques

ninodos= 125.500, TAMINODO = 128 bytes, BLOCKSIZE = 1.024 ⇒ tamAI = 15.688 bloques

Array de inodos

- Un inodo contiene las características de un directorio o fichero del sistema de ficheros:
  - **Tipo**: libre “l”, directorio “d” o fichero “f”
  - **Permisos**: lectura (r), escritura (w), ejecución (x) representados en octal (0-7):
    - 7 (111): r, w y x
    - 6 (110): r y w
    - 5 (101): r y x
    - 4 (100): r
    - 3 (011): w y x
    - 2 (010): w
    - 1 (001): x
    - 0 (000): ninguno

Array de inodos

- **Fecha y hora del último acceso a datos (atime)**
- **Fecha y hora de la última modificación de datos (mtime)**
- **Fecha y hora de la última modificación del inodo (ctime)**

  - atime, mtime y ctime son de tipo time_t
    - time_t es un long int que contiene un *timestamp* expresado en segundos después del inicio de la época UNIX (1 de Enero de 1970 00:00:00 GMT)
    - A esta forma de expresar el tiempo se la conoce como formato *epoch*
  - Se pueden inicializar con time(NULL)
  - Hay que incluir <time.h> en el fichero cabecera

Array de inodos

Cómo imprimir el formato epoch:

- función localtime (del archivo de cabecera time.h)
  - obtiene una fecha con los componentes separados (struct tm) a partir de un timestamp.
  - struct tm *localtime(const time_t *tiempoPtr);
- struct tm ;
  - La estructura contendrá al menos los siguientes miembros, en cualquier orden.
  - int tm_sec; /* los segundos después del minuto -- [0,61] */
  - int tm_min; /* los minutos después de la hora -- [0,59] */
  - int tm_hour; /* las horas desde la medianoche -- [0,23] */
  - int tm_mday; /* el día del mes -- [1,31] */
  - int tm_mon; /* los meses desde Enero -- [0,11] */
  - int tm_year; /* los años desde 1900 */
  - int tm_wday; /* los días desde el Domingo -- [0,6] */
  - int tm_yday; /* los días desde Enero -- [0,365] */
  - int tm_isdst; /* el flag del Horario de Ahorro de Energía */

Array de inodos

```
// Impresión de la fecha a partir del formato epoch
#include <stdio.h>
#include <time.h>

int main(void)
{
        time_t now;
        struct tm *ts;
        char buf[80];
        // Get current time
        time(&now);
        // Format time, "ddd yyyy-mm-dd hh:mm:ss zzz"
        ts = localtime(&now);
        strftime(buf, sizeof(buf), "%a %Y-%m-%d %H:%M:%S %Z", ts);
        printf("%s\n", buf);
        return 0;
}
```

Array de inodos

Para mostrar la información de los sellos de tiempo:

```
struct tm  *ts;
char atime[80];
char mtime[80];
char ctime[80];
struct inodo inodo;
int in;
for (in=0;in<SB.totInodos;in++) {
   inodo = leer_inodo(in);
   ts = localtime(&inodo.atime);
   strftime(atime, sizeof(atime), "%a %Y-%m-%d %H:%M:%S", ts);
   ts = localtime(&inodo.mtime);
   strftime(mtime, sizeof(mtime), "%a %Y-%m-%d %H:%M:%S", ts);
   ts = localtime(&inodo.ctime);
   strftime(ctime, sizeof(ctime), "%a %Y-%m-%d %H:%M:%S", ts);
   printf("ID: %d ATIME: %s MTIME: %s CTIME:
%s\n",in,atime,mtime,ctime);
}
```

Array de inodos

- **Cantidad de enlaces de entradas en directorio**

  Representa el total de los nombres que el archivo tiene en la jerarquía de directorios.

  Un archivo puede tener asociados diferentes nombres que correspondan a diferentes rutas, pero a través de los

  cuales accedemos a un mismo inodo y por consiguiente a los mismos bloques de datos.

  En el inodo no se especifican el (los) nombre(s) del archivo.

  - Los enlaces se crean con la función **link**
  - Ejemplo:

    Si creáis un fichero fic1 en dir1/dir2/
    y luego realizáis

    link dir1/dir2/fic1 dir3/dir4/fic2

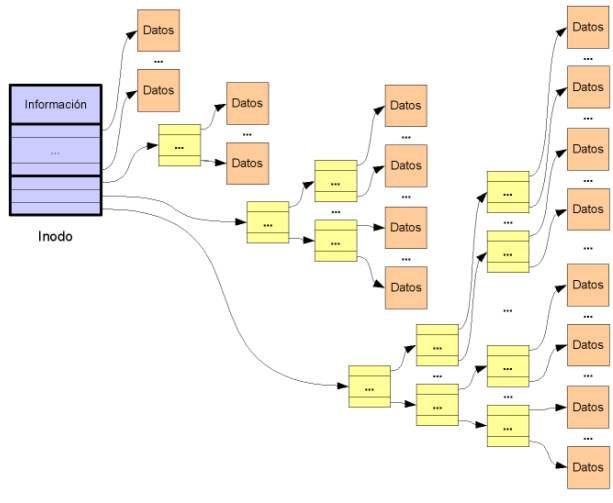
    al hacer un ls en dir3/dir4 os aparecerá fic2, cuyo contenido es el que tiene fic1

Array de inodos

- **Tamaño en bytes lógicos**
- **Cantidad de bloques ocupados en la zona de datos**
- **12 punteros a bloques directos**
  - unsigned int punterosDirectos[12]
- **3 punteros a bloques indirectos:**
  - unsigned int punterosIndirectos[3]
  - **1 puntero indirecto simple**
  - **1 puntero indirecto doble**
  - **1 puntero indirecto triple**

- Hay que observar que el nombre del archivo no queda especificado en su inodo.
  - Es en los archivos de tipo directorio donde a cada nombre de archivo se le asocia su inodo correspondiente.

Array de inodos



Tanto los bloques de datos como los bloques de punteros indirectos se almacenan en la zona de datos

Array de inodos. Gestión de inodos libres

- Dentro del array de inodos, los inodos **libres** se organizan como una **lista enlazada** (mientras que los inodos ocupados apuntan a directorios o ficheros existentes).
  - Inicialmente el 1er inodo libre es el que está en la posición 0 ⇒ se utiliza para el directorio raíz.
  - Para inicializar la lista enlazada:
    - Primeramente leeremos el superbloque para obtener la localización del array de inodos.
    - Habrá que inicializar el primer elemento del array de punteros directos de cada inodo con una variable incremental (ya que inicialmente todos los inodos están libres y en la lista enlazada cada uno apunta al siguiente).
    - El último de la lista tendrá que apuntar a un nº muy grande (NULL).
    - Iteraremos para cada bloque (desde la posición del 1er bloque del array de inodos hasta el último), y para cada inodo dentro de un bloque (cada bloque contiene una cantidad de inodos = (BLOCKSIZE/TAMINODO)).
    - El último bloque no tiene porqué estar completo.

Array de inodos. Gestión de inodos libres

```
x = 1;
para (i=SB.posPrimerBloqueAI; i<=SB.posUltimoBloqueAI; i++)
    para (j=0; j<BLOCKSIZE/TAMINODOS; j++)
        inodos[j].tipo = ”l”;        //libre
        si (x < ninodos)
            inodos[j].punterosDirectos[0] = x;
            x++;
        sino
            inodos[j].punterosDirectos[0] = UINT_MAX;
            //salir del bucle
        fsi;
    fpara;
fpara;
```

Array de inodos
Gestión de inodos libres

- Al reservar un inodo:
  - Comprobamos que quedan inodos libres
  - Inicializamos los datos de un inodo
    - Con todos los punteros a 0, nlinks=1, variables de tiempo = time(NULL)
  - Lo escribimos en el 1er inodo libre de la lista y la actualizamos

```
inodoAux = leer_inodo(SB.posPrimerInodoLibre);
ninodo = SB.posPrimerInodoLibre;
escribir_inodo(inodo, ninodo);
SB.posPrimerInodoLibre = inodoAux.punterosDirectos[0];
```

  - Decrementamos la cantidad de inodos libres

```
SB.cantInodosLibres--;
```

Array de inodos
Gestión de inodos libres

- Al liberar un inodo:
  - Liberamos todos los bloques del inodo
  - Lo marcamos como libre:

```
inodo.tipo =“l”;
```

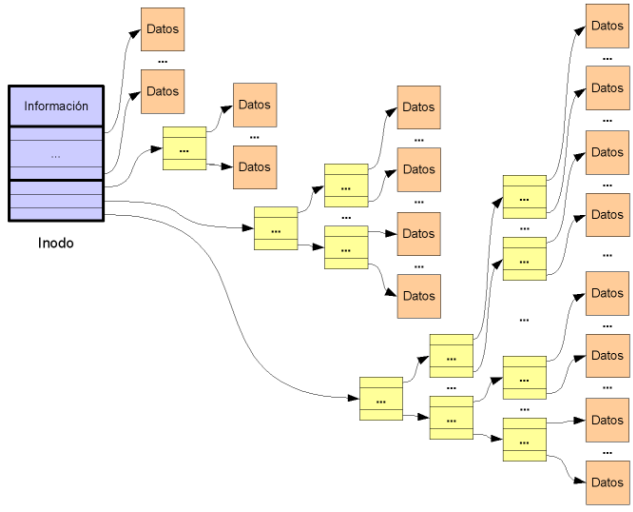
  - Lo incluimos en la lista de inodos libres por la cabecera

```
inodo.punterosDirectos[0] = SB.posPrimerInodoLibre;
SB.posPrimerInodoLibre = ninodo;
```

  - Incrementamos la cantidad de inodos libres

```
SB.cantInodosLibres++;
```

Array de inodos
Punteros a bloques directos e indirectos



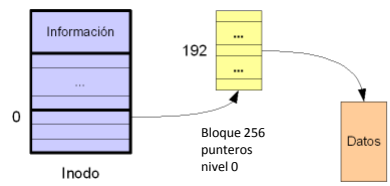
Nº bloques lógicos:

Punteros Directos:
del 0 al 11
Indirectos 0:
del 12 al 267
(12+256-1)
Indirectos 1:
del 268 al 65.803
$(12+256+256^2-1)$
Indirectos 2:
del 65.804 al
16.843.019
$(12+256+256^2+256^3-1)$
Si BLOCKSIZE=1KB ⇒
tamaño máx. ficheros:
~16GB

Array de inodos
Traducción de bloque lógico a bloque físico

Ejemplo: bloque lógico número 204

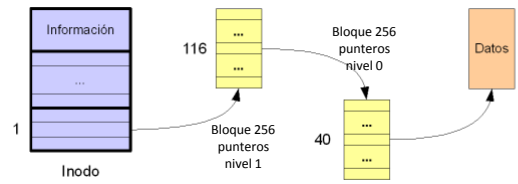
- Como 12 <= 204 <= 267, hemos de recurrir a punterosIndirectos[0].
- Restamos los punteros anteriores: 204 - 12 = **192**
- El número de nuestro bloque físico se encuentra en el puntero número **192** del bloque apuntado por punterosIndirectos[0].



Array de inodos
Traducción de bloque lógico a bloque físico

Ejemplo: bloque lógico número 30.004

- Como 268 <= 30.004 <= 65.803, hemos de recurrir a punterosIndirectos[1]
- Restamos los punteros anteriores: 30.004 - 268 = 29.736
  //así relativizamos a un valor de 0 a 65535 que son las posiciones que puedo direccionar con 2 niveles de punteros
- Calculamos 29.736 / 256 = **116**
  //así obtengo cuál de los 256 punteros de nivel 1 me apunta al bloque correspondiente de punteros de nivel 0
- Calculamos 29.736 % 256 = **40**.
  //así obtengo cuál de los 256 punteros de nivel 0 me apunta a los datos
- El nº de nuestro bloque físico se encuentra en el puntero número **40** del bloque apuntado por el puntero número **116** del bloque apuntado por punterosIndirectos[1].

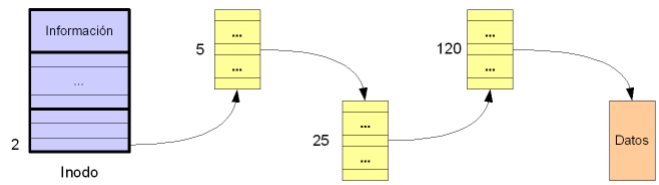


Array de inodos
Traducción de bloque lógico a bloque físico

Ejemplo: bloque lógico número 400.004

- Como 65.804 <= 400.004 <= 16.843.019, hemos de recurrir a punterosIndirectos[2].
- Calculamos 400.004 - 65.804 = 334.200
  Calculamos 334.200 / 65.536 = **5** //65.536 =256^2
  Calculamos 334.200 % 65.536 = 6.520
  Calculamos 6.520 / 256 = **25**
  Calculamos 6.520 % 256 = **120**

- El número de nuestro bloque físico se encuentra en el puntero número **120** del bloque apuntado por el puntero número **25** del bloque apuntado por el puntero número **5** del bloque apuntado por punterosIndirectos[2]..



Array de inodos
Traducción de bloque lógico a bloque físico

- int traducir_bloque_inodo (unsigned int ninodo, unsigned int blogico, unsigned int *bfisico, char reservar)
  - Si *reservar* vale 0 (por ejemplo, con *mi_read_f*), utilizaremos *traducir_bloque_inodo* únicamente para consultar:
    - Si existe bloque físico, nos devolverá su número.
    - Si no existe bloque físico, dará error.
  - Si *reservar* vale 1 (por ejemplo, con *mi_write_f*), utilizaremos *traducir_bloque_inodo* para consultar y, si no existe bloque físico, también para reservar:
    - Si existe bloque físico, nos devolverá su número.
    - Si no existe bloque físico, lo asignará y nos devolverá su número. En caso de que el bloque que estamos reservando sea de punteros, hay que inicializarlos (por ejemplo con el valor 0 que indique que no apuntan a nada).

Zona de datos

- Los directorios y ficheros en sí se almacenan en la zona de datos.
- Cada bloque de datos sólo puede ser asignado a un archivo, tanto si lo ocupa totalmente como si no.
- También contiene los índices (bloques de punteros indirectos)

Zona de datos. Directorios

- Desde el punto de vista del usuario, vamos a referenciar a los archivos mediante su *nombre* (*pathname*).
  - Un directorio no es más que un fichero especial en el que se van almacenando entradas
    - cada entrada hace referencia a un directorio o fichero contenido dentro de él.
      Podemos usar la siguiente estructura para cada entrada:

      ```
      char nombre[60];
      unsigned int inodo;    //4 bytes
      //En un bloque de 1024 bytes cabrán 16 entradas de directorio
      ```
    - Este campo *nombre* no incluye el camino de directorios (ni el carácter de separación '/').

Zona de datos. Directorios

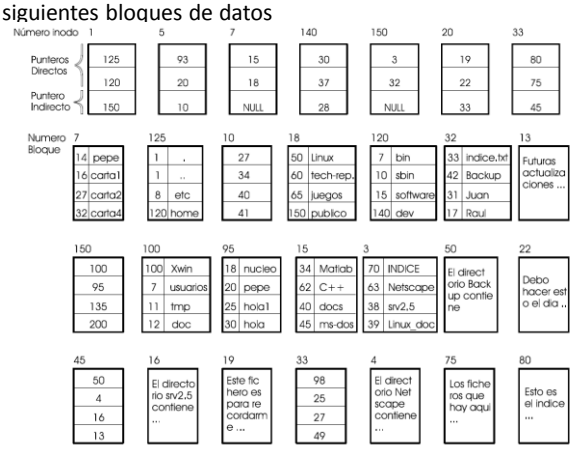
- Su función fundamental consiste en establecer la relación que existe entre el nombre de un archivo y su inodo correspondiente.
- Cuando se crea un fichero, no sólo se le tiene que asignar un inodo sino que también hay que crear una entrada de directorio.
  - Un fichero huérfano es aquél que no tiene entrada de directorio

Zona de datos.
Directorios

- Puede haber varios nombres de archivos, distribuidos por la jerarquía de directorios, que estén enlazados con un mismo inodo (enlaces).
- Si el nombre (*pathname*) es absoluto, la búsqueda del inodo del archivo se inicia en el directorio raíz
- La localización de una entrada de directorio se realiza con una búsqueda lineal (es muy costoso)

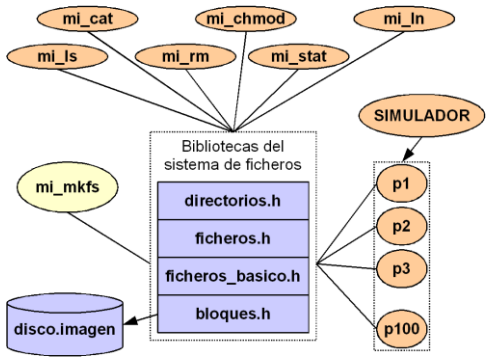
Zona de datos.
Ejercicio de localización del contenido de un fichero

- Encontrar el contenido del fichero /usuarios/publico/indice.txt en un sistema de ficheros tipo UNIX, en donde los inodos tienen dos punteros directos y uno indirecto simple, conocemos el contenido de los punteros de los siguientes inodos y el contenido de los siguientes bloques de datos



Solución animada
http://www.youtube.com/watch?v=euLJzsQwL8w

Implementación por capas



directorios.h:
funciones *extraer_camino*, *buscar_entrada*, *mi_creat*, *mi_link*, *mi_unlink*, *mi_dir*, *mi_chmod*, *mi_stat*, *mi_write*, *mi_read*

ficheros.h:
funciones *mi_write_f*, *mi_read_f*, *mi_chmod_f*, *mi_truncar_f*, *mi_stat_f*

ficheros_basico.h:
funciones *tamMB*, *tamAI*, *initSB*, *initMB*, *initAI*, escribir_bit, leer_bit, reservar_bloque, liberar_bloque, escribir_inodo, leer_inodo, reservar_inodo, liberar_inodo, liberar_bloques_inodo, traducir_bloque_inodo

bloques.h:
funciones *bmount*, *bumount*, *bwrite*, *bread*

Operación escritura en un fichero

Capa de ficheros:

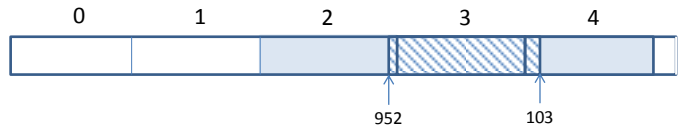
- int mi_write_f (**unsigned int *ninodo***, const void **buf_original*, unsigned int *offset*, unsigned int *nbytes*)

  Escribe el contenido de un buffer de memoria (buf_original) en un fichero/directorio (correspondiente al inodo pasado como argumento): le indicamos la posición de escritura inicial *offset* con respecto al inodo (en bytes) y el número de bytes *nbytes* que hay que escribir; **hay que devolver la cantidad de bytes escritos**.
  Esta operación sólo está permitida cuando haya permiso de escritura sobre el inodo (opción 'w'), es decir que permisos tenga el valor 010, 011, 110 o 111, lo cual se puede averiguar con la siguiente comparación: (inodo.permisos & 2) == 2.

Operación escritura en un fichero

- primerBLogico = offset/BLOCKSIZE;
- ultimoBLogico = (offset + nbytes – 1)/BLOCKSIZE;
- desp1 = offset % BLOCKSIZE; //desplazamiento en el 1er bloque
- desp2 = (offset + nbytes – 1) % BLOCKSIZE; //despl. en el último bloque

Por ejemplo, supongamos que offset = 3000 y nbytes=1200



primerBLogico = 3000/1024 = 2 ultimoBLogico = (3000+1200-1)/1024 = 4
desp1 = 3000%1024 = 952 desp2 = (3000+1200-1)%1024 = 103
escritos = (1024-952)+1024+104=1200
El resto de bytes de los bloques 2 y 4 se han de preservar!!!

void *memcpy (void *s1, const void *s2, size_t n);
// Copia los primeros **n** caracteres del objeto apuntado por **s2** al objeto apuntado por **s1**.

Operación escritura en un fichero

- Primeramente trataremos el caso en que el primer y último bloque lógicos coincidan, y por tanto el buffer que vamos a escribir (buf_original) cabe en un solo bloque:
  - Obtenemos el bloque físico a partir del bloque lógico (función traducir_bloque_inodo)
  - Leemos (función bread) el bloque físico y lo almacenamos en buf_bloque //unsigned char buf_bloque[BLOCKSIZE]
  - memcpy (buf_bloque + desp1; buf_original, nbytes);
  - Escribimos (función bwrite) el buf_bloque en el bloque físico
  - escritos = nbytes
  - Por ejemplo, supongamos que offset = 1500, nbytes = 300, BLOCKSIZE = 1024
    primerBLogico = 1500/1024 = 1 ultimoBLogico = (1500+300-1)/1024 = 1
    desp1 = 1500%1024 = 476 (desp2 = (1500+300-1)%1024 = 775)
    memcpy (buf_bloque + 476; buf_original, 300);
    bwrite(bfisico, buf_bloque):
    escritos = 300

Operación escritura en un fichero

- En caso contrario, cualquier bloque del sistema de ficheros que no vaya a ser escrito, mediante *bwrite*, en su totalidad, ha de ser previamente leído, mediante *bread*, para preservar el valor de los bytes no escritos.
- Ejemplo en el que queremos escribir 4.000 bytes (almacenados en un buffer llamado *buf_original*) a partir del byte lógico número 23.000 del inodo (suponemos que el tamaño de bloque es de 1.024 bytes):
  - El primer byte lógico que vamos a escribir es el número 23.000 (offset) Calculamos 23.000 / 1.024 = 22 (primer bloque lógico donde vamos a escribir).
  - El último byte lógico que vamos a escribir es el número 26.999 (23.000 + 4.000 – 1).
  - Calculamos 26.999 / 1.024 = 26 (último bloque lógico donde vamos a escribir).

Operación escritura en un fichero

- En caso contrario, cualquier bloque del sistema de ficheros que no vaya a ser escrito, mediante *bwrite*, en su totalidad, ha de ser previamente leído, mediante *bread*, para preservar el valor de los bytes no escritos.
- Ejemplo en el que queremos escribir 4.000 bytes (almacenados en un buffer llamado *buf_original*) a partir del byte lógico número 23.000 del inodo (suponemos que el tamaño de bloque es de 1.024 bytes):
  - El primer byte lógico que vamos a escribir es el número 23.000 (offset) Calculamos 23.000 / 1.024 = 22 (primer bloque lógico donde vamos a escribir).
  - El último byte lógico que vamos a escribir es el número 26.999 (23.000 + 4.000 – 1).
  - Calculamos 26.999 / 1.024 = 26 (último bloque lógico donde vamos a escribir).

Operación escritura en un fichero

- Distingamos tres fases:
  - Primer bloque (bloque 22):
    - Hacemos un *bread* de este bloque (con ayuda de la función *traducir_bloque_inodo*) y almacenamos el resultado en un buffer llamado *buf_bloque*.
    - Calculamos 23.000 % 1.024 = 472 //desp1
    - Ejecutamos la siguiente función:
      memcpy (buf_bloque + 472, buf_original, 1024 - 472);
    - Hacemos un *bwrite* sobre este bloque (con ayuda de la función *traducir_bloque_inodo*) con la información contenida en *buf_bloque*.
    - escritos = 1024– 472;
  - Bloques intermedios (bloques 23, 24, 25):
    - Para cada bloque lógico i:
      - Hacemos un *bwrite* (con ayuda de la función *traducir_bloque_inodo*) con la información contenida en buf_original + (1024 – 472) + (i – 22 – 1) * 1024
        escritos = escritos + 1024;

Operación escritura en un fichero

- Último bloque (bloque 26):
  - Hacemos un *bread* de este bloque (con ayuda de la función *traducir_bloque_inodo*) y almacenamos el resultado en un buffer llamado *buf_bloque*.
  - Calculamos 26.999 % 1.024 = 375 //desp2
  - Ejecutamos la siguiente función:
    memcpy (buf_bloque, buf_original + (1024 – 472) + (26 – 22 – 1) * 1024, 375 + 1);
    O memcpy (buf_bloque, buf_original + 4000 – 375 - 1, 375 + 1);
  - Hacemos un *bwrite* sobre este bloque (con ayuda de la función *traducir_bloque_inodo*) con la información contenida en *buf_bloque*.
  - escritos = escritos + 375+ 1;

Operación escritura en un fichero

- Finalmente actualizaremos la metainformación del inodo:
  - Leer el inodo
  - Actualizar el tamaño en bytes lógico **si hemos escrito más allá del final del fichero**
  - Actualizar el mtime y el ctime
  - Escribir el inodo

Operación escritura en un fichero

Capa de directorios:

- int mi_write (**const char *camino**, const void **buf*, unsigned int *offset*, unsigned int *nbytes*)
  - Buscar la entrada camino mediante la función buscar_entrada para obtener el puntero a su inodo (p_inodo).
    - Hay que comprobar si algún directorio de la ruta no tiene permiso de lectura
    - Búsqueda de coste 0(n). En realidad cada proceso tiene un directorio de trabajo para no tener que buscar desde el directorio raíz
  - Si la entrada existe llamamos a la función correspondiente de ficheros.c pasándole el p_inodo:

    mi_write_f(p_inodo, buf, offset, nbytes)

Operación lectura de un fichero

- int mi_read_f (unsigned int *ninodo*, void **buf_original*, unsigned int *offset*, unsigned int *nbytes*)
  - Lee información de un fichero/directorio (correspondiente al nº de inodo pasado como argumento) y la almacena en un buffer de memoria
  - Le indicamos la posición de lectura inicial *offset* con respecto al inodo (en bytes) y el número de bytes *nbytes* que hay que leer
  - Hay que devolver la cantidad de bytes leídos.
  - **La función no puede leer más allá del tamaño en bytes lógicos del inodo** (es decir, más allá del fin de fichero).

Operación lectura de un fichero

- Esta operación sólo está permitida cuando haya permiso de lectura sobre el inodo (opción 'r'), es decir que permisos tenga el valor 100, 101, 110 o 111, lo cual se puede averiguar con la siguiente comparación: (inodo.permisos & 4) == 4.

- La implementación es similar a mi_write_f pero permutando los 2 primeros parámetros del memcpy para el caso del primer y último bloque lógico
- Finalmente actualizaremos la metainformación del inodo (atime) y lo escribimos

Operación lectura en un fichero

Capa de directorios:

- int mi_read (const char ***camino**, void **buf*, unsigned int *offset*, unsigned int *nbytes*)
  - Buscar la entrada camino mediante la función buscar_entrada para obtener el puntero a su inodo (p_inodo).
    - Hay que comprobar si algún directorio de la ruta no tiene permiso de lectura
  - Si la entrada existe llamamos a la función correspondiente de ficheros.c pasándole el p_inodo:

    mi_read_f(p_inodo, buf, offset, nbytes)

Operación permisos (chmod)

Capa de ficheros:

- int mi_chmod_f (unsigned int ***ninodo***, unsigned char *modo*)
  - Cambia los permisos de un fichero/directorio (correspondiente al nº de inodo pasado como argumento) según indique el argumento *modo*.
  - Actualizar ctime
    - inodo.ctime=time(NULL);

Operación permisos (chmod)

Capa de directorios:

- int mi_chmod (const char ***camino***, unsigned char *modo*)

  – Buscar la entrada camino mediante la función buscar_entrada para obtener el puntero a su inodo (p_inodo).

  – Si la entrada existe llamamos a la función correspondiente de ficheros.c pasándole el p_inodo: mi_chmod_f (p_inodo, modo)

Operación metainformación (stat)

Capa de ficheros:

- int mi_stat_f (unsigned int *ninodo, struct STAT *p_stat)*
  – Devuelve la metainformación de un fichero/directorio (correspondiente al nº de inodo pasado como argumento):
    - tipo,
    - permisos,
    - cantidad de enlaces de entradas en directorio,
    - tamaño en bytes lógicos,
    - *timestamps*
    - cantidad de bloques ocupados en la zona de datos.
  – Se recomienda definir un tipo estructurado denominado STAT (podemos considerar que la estructura STAT es la misma que la estructura INODO pero sin los punteros directos e indirectos).
  – Para acceder a los campos de un *struct* pasado como puntero:

```
p_stat->tipo = inodo.tipo;
```

Operación metainformación (stat)

Capa de directorios:

- int mi_stat (const char ***camino***, STAT **p_stat*)
  - Buscar la entrada camino mediante la función buscar_entrada para obtener el puntero a su inodo (p_inodo).
  - Si la entrada existe llamamos a la función correspodiente de ficheros.c pasándole el p_inodo: mi_stat_f (p_inodo, pstat)

Operación truncar

Capa de ficheros:

- int mi_truncar_f (unsigned int n*inodo*, unsigned int *nbytes*);
  - Trunca un fichero/directorio (correspondiente al nº de inodo pasado como argumento) a los bytes indicados, liberando los bloques que no hagan falta
    - Si se trunca a 0 bytes, hay que liberar todos los bloques
  - Se basa en la función *liberar_bloques_inodo* desarrollada para la función *liberar_inodo*.
  - Hay que comprobar que el inodo tenga permisos de escritura.
  - Actualizar mtime, ctime y el tamaño en bytes lógicos del inodo (pasará a ser igual a nbytes)!

Operación truncar

Capa de ficheros:

- int mi_truncar_f (unsigned int n*inodo*, unsigned int *nbytes*);
  - Trunca un fichero/directorio (correspondiente al nº de inodo pasado como argumento) a los bytes indicados, liberando los bloques que no hagan falta
    - Si se trunca a 0 bytes, hay que liberar todos los bloques
  - Se basa en la función *liberar_bloques_inodo* desarrollada para la función *liberar_inodo*.
  - Hay que comprobar que el inodo tenga permisos de escritura.
  - Actualizar mtime, ctime y el tamaño en bytes lógicos del inodo (pasará a ser igual a nbytes)!

Concurrencia

- No es responsabilidad del sistema de ficheros controlar el acceso concurrente a los bloques de datos, sino de aquellos programas cliente que así lo deseen.
- Sí hay que controlar, mediante secciones críticas, inconsistencias en los metadatos (superbloque, mapa de bits y array de inodos) para evitar que
  - No se acceda correctamente a los bloques de datos
  - Se creen archivos huérfanos
  - Se incurra en una pérdida de privacidad
- Esta responsabilidad es del sistema de ficheros, nunca de los programas cliente.

Concurrencia

- Pueden surgir problemas de tipo físico, como cortes eléctricos
- Recuperación:
  - Hace años se utilizaba un algoritmo de recuperación de coste $O(n^2)$:
    - "scan disk" en Windows 95
    - "FSCK" en Linux
  - journaling (coste O(1))
    - Lleva cuenta de las operaciones pendientes, lo almacena en memoria flash
  - RAID
    - Duplica todo en 2 discos, aunque ralentiza las operaciones

Concurrencia

- **Funciones de directorios.c** que requieren semáforo:
  - **mi_creat()**
    - 2 procesos que lo utilizasen simultáneamente podrían asignar el mismo inodo al fichero, buscar los 2 el 1er inodo libre al mismo tiempo
  - **mi_link()**
  - **mi_unlink()**
  - **mi_write()**

- En **mi_dir()**, **mi_chmod()**, **mi_stat()** y **mi_read()** no hace falta semáforo.

Concurrencia

- No ponemos semáforo en el **mi_read()**
  - es la función más frecuentemente llamada
  - si se entra en la sección crítica para cada llamada, sólo un proceso podría leer un fichero, los demás deberían esperar (**serialización**).
  - Pero se ha de ir con cuidado en el orden de las otras funciones!!!. Por ejemplo:
    - eliminar la entrada de directorio antes de eliminar el inodo
    - marcar los punteros del inodo como "null" antes de liberar los bloques
    - grabar 1º los bloques antes de modificar el tamaño del fichero

Concurrencia

- El control de consistencia de ***mi_write*()** se puede realizar a través de la función **mi_write_f()** de la capa de ficheros.
  - El semáforo es necesario **sólo cuando el fichero crece**, no es preciso serializar toda la escritura.

Concurrencia

Si se desea una mayor granularidad, las secciones críticas que tendremos que controlar serán las porciones de código donde se **reserven** o **liberen bloques** o **inodos**:

- **ficheros_basico.c**:
  - ***traducir_bloque_inodo*** (sólo si reserva un bloque). Permite el control de la concurrencia al ejecutar las funciones *mi_creat*, *mi_write* y *mi_link*.
- **ficheros.c**:
  - ***mi_truncar_f*** (por liberar bloques). Permite el control parcial de la concurrencia al ejecutar la función *mi_unlink*.
- **directorios.c**:
  - ***buscar_entrada*** (sólo si reserva un inodo). Permite el control de la concurrencia al ejecutar la función *mi_creat*
  - ***mi_unlink*** (por liberar un inodo y sus bloques)