
UNIVERSIDAD DE GRANADA

Sistemas Operativos

# Tarea 3.1

*Entrega de ejercicios del tema 4*

*Javier Sáez de la Coba*

Curso 2017-2018

4 de enero de 2018

# 1. Ejercicio 2

## Enunciado

Si se pierde el primer puntero de la lista de espacio libre, ¿podría el Sistema Operativo reconstruirla? ¿Cómo?

## Solución

Si por cualquier motivo se perdiera la lista de espacio libre y fuera imposible reconstruirla (mediante copias de seguridad o RAID), el sistema podría rehacerla escaneando todo el disco y comparando los sectores ocupados por los archivos según la estructura del SA. Una vez escaneado, el Sistema de Archivos tendría constancia de qué bloques están ocupados y cuales no, pudiendo almacenarlos en una nueva lista de espacio libre.

# 2. Ejercicio 3

## Enunciado

El espacio libre en disco puede ser implementado usando una lista encadenada con agrupación o un mapa de bits. La dirección en disco requiere D bits. Sea un disco con B bloques, en que F están libres. ¿En qué condición la lista usa menos espacio que el mapa de bits?

## Solución

El mapa de bits necesita exactamente $B$ bits para almacenarse, al requerir un bit por cada bloque en disco.

Una lista enlazada con agrupación puede guardar $n$ direcciones a disco de $D$ bits cada una en cada bloque de la lista. La $n$-ésima dirección apunta al siguiente bloque de la lista. Por ello, el tamaño que ocupa la lista enlazada con agrupación es: $F/(n-1) * D + D$.

Calibrando el parámetro $n$ y en el momento que el espacio libre decrezca puede darse el caso de que la lista enlazada agrupada ocupe menos que el mapa de bits.

# 3. Ejercicio 6

## Enunciado

Un i-nodo del sistemas de archivos *s5fs* de UNIX tiene 10 direcciones de disco para los diez primeros bloques de datos, y tres direcciones más para realizar una indexación a uno, dos y tres niveles. Si cada bloque índice tiene 256 direcciones de bloques de disco ¿cuál es el tamaño del mayor archivo que puede ser manejado, suponiendo que 1 bloque de disco es de 1KByte?

## Solución

Si cada bloque índice puede almacenar 256 direcciones, entonces, podemos direccionar:

$10$ bloques $+2^8$ bloques (primer nivel) $+2^8 * 2^8 (= 2^{16})$ bloques (segundo nivel) $+2^{16} * 2^8 (= 2^{24})$ bloques (tercer nivel) $= 16843018$ bloques direccionables

$16843018 \approx 2^{24} \rightarrow 2^{24} * 2^{10}$ bytes direccionables. $\rightarrow 2^4 * 2^{30}$ bytes. Por lo que el tamaño máximo de cada archivo sería **16GB** (realmente un poco más debido a la aproximación que hemos hecho antes.)

# 4. Ejercicio 7

## Enunciado

Sobre conversión de direcciones lógicas dentro de un archivo a direcciones físicas de disco. Estamos utilizando la estrategia de indexación a tres niveles para asignar espacio en disco. Tenemos que el tamaño de bloque es igual a 512 bytes, y el tamaño de puntero es de 4 bytes. Se recibe la solicitud por parte de un proceso de usuario de leer el carácter número N de determinado archivo. Suponemos que ya hemos leído la entrada del directorio asociada a este archivo, es decir, tenemos en memoria los datos PRIMER-BLOQUE y TAMAÑO. Calcule la sucesión de direcciones de bloque que se leen hasta llegar al bloque de datos que posee el citado carácter.

## Solución

Por cada bloque en disco podemos guardar: $512//4 \ = \ 128$ direcciones a disco ($//$ hace referencia al cociente de la división entera). Para leer el caracter (1 byte) N de un fichero:

dividimos $B_d = N//512$ para saber el bloque en el que se encuentra dicho caracter. Una vez tenemos el número de bloque:

- Si es bloque es uno de los 10 primeros (bloques del 0 al 9), accedemos directamente al bloque, ya que su dirección está guardada en el i-nodo que ya está cargado en memoria principal.
- Si **no** está en entre los 10 primeros bloques: Comprobamos en que nivel de indexación está:
  Nivel de indexación 1: bloques desde el 10 hasta el 137
  - Leemos el bloque apuntado por i-nodo[10], el índice de nivel 1, que contiene 128 punteros a direcciones de disco correspondientes a los bloques 10...137 del archivo. Se lee el bloque de datos apuntado por $i = B_d - 10$, es decir, el bloque i-nodo[10][$i$].

  Nivel de indexación 2: bloques desde el 138 hasta el 16521
  - Leemos el bloque apuntado por i-nodo[11], el índice de nivel 2, que contiene 128 punteros a 128 subíndices. Estos subíndices contienen a su vez 128 punteros a bloques de datos del disco. Así el primer subíndice apunta a los bloques [138...265] del archivo, el segundo subíndice a los bloques [266...393]... Para saber a que subíndice debemos acceder, hay que leer el subíndice que se encuentra en la posición $i = (B_d - 138)//128$, leyendo por tanto el bloque de disco i-nodo[11][$i$]. Después, se lee dentro de este subíndice el bloque de datos deseado que se encuentra en la posición $j = (B_d - 138 - i * 128)//128$, leyendo por tanto el bloque de disco i-nodo[11][$i$][$j$]

  Nivel de indexación 3: bloques desde el 16522 hasta el 2113673 (tamaño máximo)
  - Leemos el bloque apuntado por i-nodo[12], el índice de nivel 3, que contiene 128 punteros a 128 subíndices. Estos subíndices contienen a su vez otros 128 punteros a otros 128 subíndices. En estos últimos se encuentran 128 punteros a bloques de datos del archivo. Por tanto hay que leer el primer subíndice que se encuentra en la posición $i = B_d - 16522//16384$ del índice de nivel 3. Después, se lee dentro de este subíndice el siguiente subíndice (correspondiente a un índice de nivel 1) que se encuentra en la posición $j = (B_d - 16522 - i * 16384)//128$. Ahora se lee en este subíndice la dirección del bloque de datos que queremos leer, que se encuentra en la posición $k = (B_d - 16522 - i * 16384) - (j * 128)$, leyendo así el bloque de datos apuntado por i-nodo[12][$i$][$j$][$k$]

En resumen, la cadena de lecturas correspondiente a los distintos niveles de indexación son:

- Nivel 0 (acceso directo):
  bloque apuntado por el array del i-nodo

- Nivel 1 (indexación simple)
  bloque 10 (índice) → bloque de datos
- Nivel 2 (indexación doble)
  bloque 11 (índice1) → índice2 → bloque de datos
- Nivel 3 (indexación triple)
  bloque 12 (índice1) → índice2 → índice3 → bloque de datos

Aquí hay un fragmento de código en python que hace todos los cálculos necesarios de forma genérica (basándose en los parámetros tamaño de bloque y tamaño de puntero) y muestra por pantalla los bloques que hay que leer para acceder al caracter deseado.

```
#!/usr/bin/python3

EN_INODO = 10

caracter_deseado = int(input("Introduce el numero de caracter del
    archivo que deseas: "))
tam_bloque = int(input("Introduce el tamaño de bloques (en bytes)
   : "))
tam_direccion = int(input("Introduce el tamaño de un puntero a
   disco (en bytes): "))

dir_bloque = tam_bloque // tam_direccion
bloque_deseado = caracter_deseado // tam_bloque
#bloque_deseado = int(input("Introduce el bloque deseado: "))


def calcula_indice_nivel1(deseado, base):
    resultado = deseado-base
    return resultado

def calcula_indice_nivel2(deseado, base):
    deseado_relativo = deseado-base
    indice1 = deseado_relativo // dir_bloque
    indice2 = calcula_indice_nivel1(deseado_relativo, indice1*
       dir_bloque)

    return [indice1, indice2]

def calcula_indice_nivel3(deseado, base):
    deseado_relativo = deseado-base
    indice1 = deseado_relativo // (dir_bloque**2)
```

```
    resultado = [indice1]
    return(resultado + calcula_indice_nivel2(deseado_relativo,
       indice1*dir_bloque*dir_bloque))


base_nivel0 = 0
tope_nivel0 = EN_INODO - 1
base_nivel1 = tope_nivel0 + 1
tope_nivel1 = dir_bloque + base_nivel1 - 1
if(bloque_deseado <= tope_nivel0):
    print("El bloque es de acceso directo, su direccion está en
       el i-nodo")
else:

    if(bloque_deseado <= (tope_nivel1)):
        indicen1 = calcula_indice_nivel1(bloque_deseado,10)
        print("Está en la indexacion de nivel 1.")
        print("Leer i-nodo["+str(EN_INODO)+"]["+ str(indicen1) +
           "]")
    else:
        base_nivel2 = tope_nivel1 + 1
        tope_nivel2 = dir_bloque * dir_bloque + base_nivel2 - 1
        if (bloque_deseado <= tope_nivel2):
            indice2 = calcula_indice_nivel2(bloque_deseado,
               base_nivel2)
            print("Está en la indexación de nivel 2. \n")
            print("Leer: i-nodo["+str(EN_INODO+1)+"]["+str(
               indice2[0])+"][" +str(indice2[1])+ "]")
        else:
            base_nivel3 = tope_nivel2 + 1
            tope_nivel3 = base_nivel3 + dir_bloque**3 - 1
            # Comprobar si se excede el tamaño maximo del archivo
            if (bloque_deseado > tope_nivel3):
                print("El archivo excede los limites del
                   almacenamiento")
                raise SystemExit
            # Si no excede los limites:
            indice3 = calcula_indice_nivel3(bloque_deseado,
               base_nivel3)
            print("Está en la indexación de nivel 3.\n")
            print("Leer: i-nodo["+str(EN_INODO+2)+"]["+str(
               indice3[0])+"]["+str(indice3[1])+"]["+str(indice3
               [2])+"]")
```

Ejemplo de salida:

```
~$ python3 calcular_bloque.py
Introduce el numero de caracter del archivo que deseas: 67876645
Introduce el tamaño de bloques (en bytes): 512
Introduce el tamaño de un puntero a disco (en bytes): 4
Está en la indexación de nivel 3.
Leer: i-nodo[12][7][10][81]
```

# 5. Ejercicio 12

## Enunciado

Para comprobar la consistencia de un sistema de archivos de Unix, el comprobador de consistencia (fsck) construye dos listas de contadores (cada contador mantiene información de un bloque de disco), la primera lista registra si el bloque está asignado a algún archivo y la segunda, si está libre. Según se muestra en la siguiente figura:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| en uso: | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| libres: | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |

¿Existen errores? ¿Son serios estos errores? ¿por qué? ¿qué acciones correctivas sería necesario realizar sobre la información del sistema de archivos?

## Solución

**Si** existen errores según estos contadores. Esto se ve en que los contadores no son contrarios (uno la negación del otro). Hay dos tipos de problemas, uno más serio que el otro. Estos son:

- Ambos contadores a **0**
  Esto implica que el bloque no está en uso pero tampoco está marcado como libre, por lo que ese espacio no se aprovecha.
- Ambos contadores a **1**
  Esto implica que el bloque se supone en uso por un archivo pero que está marcado como libre, con el consiguiente riesgo de sobreescribir los datos y corromper los archivos.

Para el primer problema, lo más sencillo es marcar este bloque como libre, ya que no está marcado como usado por ningún i-nodo. Para el segundo problema, hay dos posibles causas. O un archivo no se ha terminado de borrar (para lo que convendría revisar el `journal` del sistema de archivos. O la lista de espacio libre ha sufrido un problema. Si no se puede recuperar la transacción perdida (y borrar el archivo completamente), lo más seguro es marcar el bloque como no libre, ya que cuando el i-nodo que lo tiene marcado como usado se elimine, el bloque volverá a marcarse como libre.

# 6. Ejercicio 16

## Enunciado

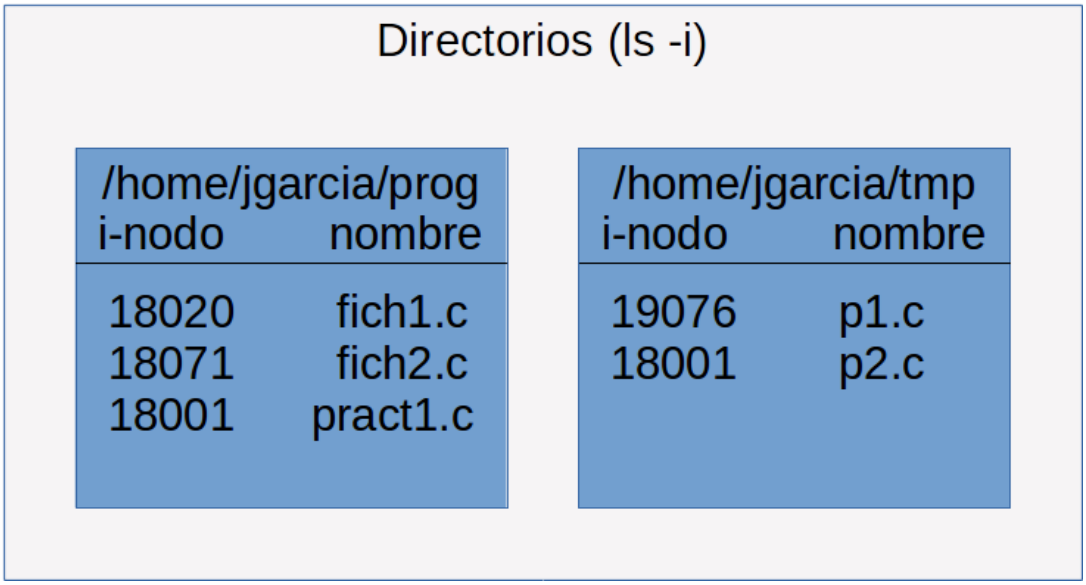
Suponiendo una ejecución correcta de las siguientes órdenes en el sistema operativo Unix:

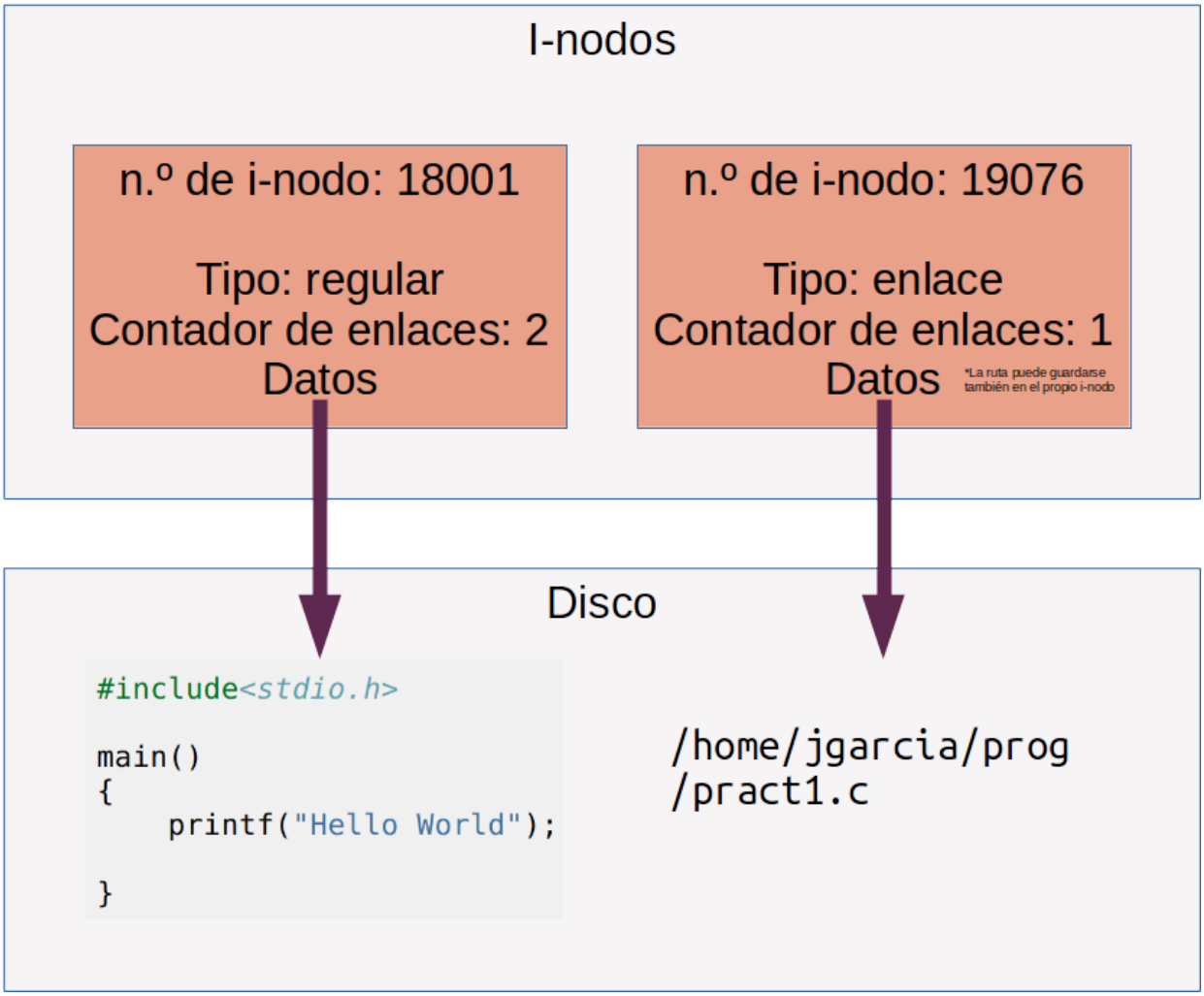
```
/home/jgarcia/prog > ls -i  (* lista los archivos y sus números de i-nodos del
18020 fich1.c                  directorio prog*)
18071 fich2.c
18001 pract1.c

/home/jgarcia/prog > cd ../tmp
/home/jgarcia/tmp > ln -s ../prog/pract1.c p1.c (* crea un enlace simbólico *)
/home/jgarcia/tmp > ln ../prog/pract1.c p2.c (* crea un enlace absoluto o duro*)
```

represente gráficamente cómo y dónde quedaría reflejada y almacenada toda la información referente a la creación anterior de un enlace simbólico y absoluto (”hard”) a un mismo archivo, pract1.c.

**Solución**


Directorios (ls -i)
/home/jgarcia/prog
i-nodo nombre
18020 fich1.c
18071 fich2.c
18001 pract1.c
/home/jgarcia/tmp
i-nodo nombre
19076 p1.c
18001 p2.c


I-nodos
n.º de i-nodo: 18001
Tipo: regular
Contador de enlaces: 2
Datos
n.º de i-nodo: 19076
Tipo: enlace
Contador de enlaces: 1
Datos
*La ruta puede guardarse también en el propio i-nodo
Disco
#include<stdio.h>
main()
{
printf("Hello World");
}
/home/jgarcia/prog
/pract1.c

# 7. Ejercicio 22

## Enunciado

¿Cómo consigue optimizar FFS operaciones del tipo “ls –l” sobre un directorio? Explíquelo y ponga un ejemplo.

## Solución

FFS optimiza las operaciones de listado de archivos de un directorio gracias a su política de asignación. Dicha política coloca los archivos de un mismo directorio en un grupo del sistema de archivos. Esta organización de grupos del sistema de archivos mejora la eficiencia en las búsquedas de archivos, ya que cada grupo mantiene su propia tabla de i-nodos, por lo que hay menos inodos en los que buscar, acelerando la recuperación de la información de los i-nodos correspondientes a un mismo directorio.

# 8. Ejercicio 24

## Enunciado

¿Es correcto el siguiente programa en UNIX, en el sentido de que no genera inconsistencias en las estructuras de datos que representan el estado de los archivos en el sistema? Represente con un esquema gráfico la información almacenada en las estructuras (tablas de descriptores de archivos, tabla de archivos y tabla de i-nodos) que representan el estado real referente a los archivos utilizados por los siguientes procesos en el instante de tiempo en que: el PROCESO HIJO DE A ha ejecutado la instrucción marcada con *(1)*, el PADRE (*PROCESO A*) ha ejecutado la *(2)*, y el PROCESO B ha ejecutado la *(3)*. Para ello suponga que son los únicos procesos ejecutándose en el sistema en ese momento.

```
/* PROCESO A */
main(){
        int archivos[2], pid;
        archivos[0]=open("/users/prog.c",O_RDONLY);
        archivos[1]=open("/users/prog.c",O_RDWR);
        if ((pid=fork())==-1)
                perror("Ha ocurrido un error en la creación del
                   proceso");
        else if (pid==0)
                execlp ("ls", "ls", NULL); /*(1)*/
        else
                close(archivos[0]); /*(2)*/
}
/* PROCESO B */
main(){
        int archivos[2];
        archivos[0]=open("/users/prog.c",O_RDONLY);
        archivos[1]=open("/users/texto.txt",O_RDWR);
        dup (archivos[0]); /* (3) */
}
```

## Solución

El código es correcto, pues no genera inconsistencias. Hay un fallo a la hora de representar los archivos abiertos por el proceso hijo de A, pues al ejecutar "ls", se abren archivos y directorios dependientes del entorno de ejecución, el cual no se ha especificado.

Tablas de descriptores de archivos