

# MEMORIA DE PRÁCTICAS



## **Ampliación de Sistemas Operativos**

María Dolores Guerrero Munuera

23054800Z

María Terea Moreno Boluda

49171283C

Profesor: Diego Sevilla Ruiz

Murcia

Convocatoria de Enero

## **Contenido**

<b>1. Introducción.....</b>	<b>3</b>
<b>2. Boletín VII.....</b>	<b>3</b>
<b>2.1. Syscall date.....</b>	<b>3</b>
<b>2.2. Syscall dup2.....</b>	<b>6</b>
<b>3. Boletín VIII.....</b>	<b>9</b>
<b>3.1. Ejercicio I.....</b>	<b>9</b>
.....	9
<b>3.2. Ejercicio II.....</b>	<b>9</b>
<b>3.2.1. Argumento negativo.....</b>	<b>9</b>
<b>3.2.2. Manejo de fallos en la página inválida debajo de la pila.....</b>	<b>10</b>
<b>3.2.3. Verificación del correcto funcionamiento de fork(), exit() y wait().....</b>	<b>12</b>
<b>3.2.4. Uso correcto del kernel de páginas de usuario sin reservar.....</b>	<b>13</b>
<b>4. Boletín IX.....</b>	<b>14</b>
<b>4.1. Ejercicio I.....</b>	<b>14</b>
<b>4.2. Ejercicio II.....</b>	<b>17</b>

# 1. Introducción

A lo largo del siguiente documento, se va a llevar a cabo la explicación en detalle de todos los boletines de prácticas resueltos durante la segunda parte de las prácticas de la asignatura de Ampliación de Sistemas Operativos.

Este bloque se centra en la realización de una serie de cambios sobre el software xv6 que consiste en un Sistema Operativo a pequeña escala con una arquitectura 8086.

## 2. Boletín VII

El primer boletín de este bloque propone la implementación de dos llamadas al sistema. Por un lado, el caso de ejemplo de la implementación de la syscall *date* y por otro la implementación de *dup2*.

### 2.1. Syscall date

El primer paso a seguir para realizar este apartado, ha sido la creación de un programa de prueba que se puede observar en la *Figura 2:0*.

```
1  #include "types.h"
2  #include "user.h"
3  #include "date.h"
4
5  int
6  main ( int argc , char * argv []){
7      struct rtcdate r;
8      if(date(&r)){ //si date devuelve error
9          printf(2, "date failed\n");
10         exit();
11     }
12     printf(1,"La hora actual es: %d:%d:%d\n", r.hour,r.minute,r.second);
13     exit();
14 }
```

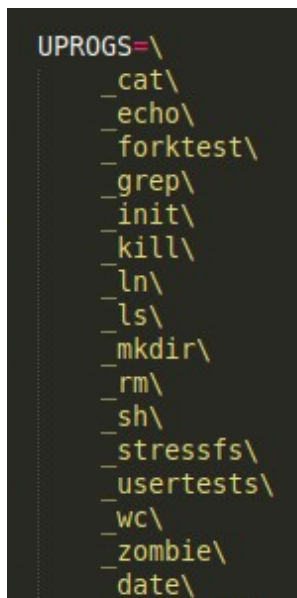
Figura 2:0 Programa de prueba para la llamada al sistema date

Este programa nos ayudará a probar la implementación de nuestra llamada al sistema *date*. Su ejecución es la siguiente: en la variable *r*, cuyo tipo es la estructura *rtcdate*, se almacenará el momento actual. Esto se consigue mediante la llamada al sistema *date*.

Si la llamada al sistema no produce un error, se mostrará por la salida estándar la hora actual, con el formato *hh:mm:ss*. En caso de error, se mostrará el mensaje de error *date failed*.

Para la implementación de la llamada al sistema *date* se han seguido una serie de pasos explicados a continuación:

- Se ha incluido, en el fichero *Makefile*, en la sección *UPROGS* la referencia al programa de prueba tal y como se puede ver en la *Figura 2:1*.



```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
_init\
_kill\
_ln\
_ls\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_zombie\
_date\
```

Figura 2:1

- En el fichero *syscall.h*, se ha incluido un nuevo identificador de llamada al sistema para la llamada *SYS\_date* con la siguiente línea de código:

```
#define SYS_date 22
```

- En el fichero *usys.S*, se ha añadido la nueva llamada con la siguiente línea de código:

```
SYSCALL(date)
```

- En el fichero *syscall.c*, se ha implementado la definición de la función *sys\_date()* incluyendo la declaración de la función que ejecuta la llamada al sistema y su identificador de llamada en el array correspondiente a través de la inclusión de las siguientes líneas de código:

```
extern int sys_date(void);

[SYS_date] sys_date
```

- En el fichero `sysproc.c`, se incluirá la nueva llamada, `sys_date()`, cuya implementación podemos observar en la *Figura 2:2*.

```
93 //implementación de la llamada al sistema date
94 int
95 sys_date(void){
96     struct rtcdate *r;
97     if(argptr(0, (void*)&r, sizeof(*r))<0)
98         return -1;
99     cmostime(r);
100     return 0;
101 }
```

Figura 2:2

Como se puede observar, en primer lugar obtenemos el puntero que se nos pasa como parámetro para guardar el resultado y lo almacenamos en la variable `r`. En caso de producirse un error al recuperar el argumento, la llamada retornará el valor `-1`. En caso de que no haya problemas, se invocará a la función `cmostime` que será la encargada de devolver el instante actual y de guardar esa información en la dirección de memoria pasada como parámetro.

- Por último, se añadirá la definición de `date()` al fichero `user.h` para poder utilizar la implementación anterior, con la siguiente línea de código:

```
int date(struct rtcdate *d)
```

- Para verificar que el desarrollo de esta llamada al sistema funciona, se incluye un ejemplo de prueba en la *Figura 2:3*.

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ date
La hora actual es: 10:36:33
$
```

Figura 2:3

## 2.2. Syscall dup2

La implementación de esta llamada al sistema, *dup2*, es similar a la realizada anteriormente. Para su desarrollo se ha creado un nuevo programa de prueba con el fin de poder comprobar el comportamiento de *dup2*. Este programa de prueba incluirá la comprobación de los casos con descriptores de fichero erróneos, no mapeados y syscalls correctas.

El objetivo de esta llamada al sistema es el de, dados dos descriptores de ficheros, asociar el fichero abierto al que apunta el inicial al nuevo descriptor, siempre y cuando se cumplan las comprobaciones de seguridad.

A continuación se va a detallar la forma en que se ha implementado la llamada al sistema:

- En el fichero *syscall.h*, se ha añadido un nuevo identificador para nuestra llamada con la siguiente línea de código:

```
#define SYS_dup2 23
```

- En el fichero *usys.S*, hemos añadido la siguiente línea:

```
SYSCALL(dup2)
```

Este paso es muy importante, ya que si no se incluyese esto, xv6 no tendría definida una regla para la llamada al sistema *dup2* y, por consiguiente, el código completo del Sistema Operativo no compilaría.

- En el fichero *syscall.c*, se incluirá la definición de la nueva llamada al sistema a través de las siguientes líneas:

```
extern int sys_dup2(void);  
[SYS_dup2] sys_dup2;
```

En este caso, se ha implementado la llamada al sistema dentro del fichero *sysfile.c* en lugar de utilizar *sysproc.c* como en la anterior implementación. Esto se debe a que en el actual fichero encontramos todas las llamadas al sistema necesarias para el tratamiento de ficheros. En la *Figura 2:4*, podemos observar el resultado final de la implementación.

```

70 int
71 sys_dup2(void){
72     //dup2(fd1,fd2) fd1 el q se quiere duplicar fd2 al nuevo
73     struct file *f;
74     int oldfd, newfd;
75
76     // Se cogen los descriptors de ficheros y el fichero a duplicar.
77     //obtener el primer argumento usando argfd si hay error -1
78     if(argfd(0, &oldfd, &f) < 0)
79         return -1;
80     //Obtener el segundo argumento
81     if (argint(1,&newfd) < 0) //si no es un entero
82         return -1;
83
84     if(newfd < 0 || newfd >= NOFILE) //si no es válido
85         return -1;
86
87     //si los dos descriptors son iguales no se hace nada
88     // Si ambos descriptors coinciden, no hacemos nada.
89     if (oldfd == newfd)
90         return newfd;
91     //si no son iguales
92     //1. comprobar si fd2 está abierto, lo cerramos.
93     if(myproc()->ofile[newfd] != 0)
94         fileclose(myproc()->ofile[newfd]);
95
96     //2. duplicar fd1 en fd2
97     myproc()->ofile[newfd]=filedup(f); //en f se guarda el id de la pila de ficheros de oldfd
98
99     return newfd;
100
101 }
102

```

Figura 2:4

Como se puede observar, lo primero que se realiza son comprobaciones de error en los dos descriptors de fichero. Tras esto, se comprueba que los dos descriptors no sean iguales, ya que, si lo fuesen, no se deberá de realizar ninguna operación. Si no son iguales se deberá comprobar si el fichero al que se quiere cambiar su descriptor está abierto, si esto es así se cerrará y acto seguido se duplicará el descriptor de fichero.

- Se añadirá la definición de la nueva llamada al sistema en el fichero *user.h* con la siguiente línea de código:

*int dup2(int,int)*

- Para poder utilizar el programa implementado, se debe incluir en la sección *UPROGS* del fichero *Makefile* el nombre del programa, como se puede observar en la Figura 2:5.

```

167
168 UPROGS=\
169     _cat\
170     _echo\
171     _forktest\
172     _grep\
173     _init\
174     _kill\
175     _ln\
176     _ls\
177     _mkdir\
178     _rm\
179     _sh\
180     _stressfs\
181     _usertests\
182     _wc\
183     _zombie\
184     _date\
185     _dup2test\
186

```

Figura 2:5

Para verificar que la implementación funciona, se muestra, en la *Figura 2:6*, la salida del programa de prueba.

```

QEMU
SeaBIOS (version 1.10.2-1ubuntu1)

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF8DDDD+1FECDDDD C980

Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
U$ dup2test
Este mensaje debe salir por terminal.
Este mensaje debe salir por terminal.
$ _

```

Figura 2:6



## 3. Boletín VIII

A lo largo del siguiente apartado, se va a realizar la explicación del código que se ha desarrollado para el manejo de reserva de páginas bajo demanda.

### 3.1. Ejercicio I

Para la implementación de este tipo de reserva tenemos que modificar la llamada al sistema, `sbrk()`, implementada en la función `sys_sbrk()` del fichero `sysproc.c`. En esta función se ha sustituido la llamada a la función `growproc()` por un incremento del tamaño de proceso. En la Figura 3:0 podemos ver las modificaciones realizadas.

```
int
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0) //obtiene el primer argumento n
        return -1;

    addr = myproc()->sz;
    myproc()->sz += n; //modificar tam.
    return addr;
}
```

Figura 3:0

Ahora cuando se intente reservar memoria, esto provocará un fallo, por lo que tenemos que capturar y tratar dicho fallo con el manejador y reservar la memoria en dicho manejador. Para ello, añadimos una nueva entrada, `T_PGFLT`, en el switch de la función `trap()` del fichero `trap.c` donde se llevará a cabo un mapeo de una nueva página física en la dirección que generó el fallo.

### 3.2. Ejercicio II

Dado que la implementación anterior no es totalmente correcta, ya que no contempla algunos casos de error. Esto será lo que se implementará en este ejercicio.

#### 3.2.1. Argumento negativo

Cuando a la función `sys_sbrk()` se le pasa como parámetro un número negativo, libera memoria. Para esta implementación se ha de modificado la función `sys_sbrk()` mencionada en el ejercicio anterior, Figura 3:1.

```

int
sys_sbrk(void)
{
    int addr;
    int n;

    if(argint(0, &n) < 0) //obtiene el primer argumento n
        return -1;

    addr = myproc()->sz;

    if(n < 0){ //arg. negativo
        growproc(n);
    }else{
        myproc()->sz += n; //modificar tam.
    }

    return addr;
}

```

Figura 3:1

Antes de modificar el tamaño del proceso se comprueba si el argumento que se le ha pasado como parámetro es negativo. En ese caso, se realiza la llamada a la función *growproc()*, ya que ésta ya realiza la comprobación y cuando el parámetro que se le pasa es un número negativo, libera páginas de memoria y actualiza la tabla de páginas, con la ayuda de la función *deallocvm()*.

Para la comprobación del correcto funcionamiento de esta función se ha hecho uso del programa de prueba *tsbrk1.c* proporcionado por los profesores de la asignatura. Como podemos ver en la Figura 3:2, el funcionamiento es el esperado.

```

Booting from Hard Disk...
cpu0: starting 0
sb: size 20000 nblocks 19937 ninodes 200 nlog 30 logstart 2
tart 58
init: starting sh
$ tsbrk1
Debe imprimir 1: 1.
$

```

Figura 3:2

### 3.2.2. Manejo de fallos en la página inválida debajo de la pila

Para esta implementación se ha añadido un atributo a la estructura *proc*, declarada en el fichero *proc.h*, para guardar la dirección donde comienza la página inválida, dicho atributo ha sido declarado con el nombre de *initPila* y ha sido inicializado en el fichero *exec.c* en la función *exec()*. Esto se puede ver en la Figura 3:3.

```

// Allocate two pages at the next page boundary.
// Make the first inaccessible. Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

//inicialización dir. para b8-pila
curproc->initPila = sp-PGSIZE; //resto tam pág.

```

Figura 3:3

Una vez realizada la inicialización, se ha modificado la función trap, Figura 3:4, situada en el fichero trap.c para comprobar si el nuevo tamaño de página está por debajo del valor de *initPila*, si esta condición se cumple, supondrá un desbordamiento de página, por lo que saltará un mensaje de error informando de esto.

```

//asignación de memoria bajo demanda
uint a = PGROUNDDOWN(rcr2());
/*comprobar fallos en la página inválida debajo de la pila*/
//si el nuevo tamaño de página está por debajo de la pila
if (a <= myproc()->initPila) {
    cprintf("Desbordamiento de pila\n");
    myproc()->killed = 1;
    break;
}

```

Figura 3:4

Para la comprobación del correcto funcionamiento de la nueva implementación se ha hecho uso del programa de prueba tsbrk2.c proporcionado por los profesores de la asignatura. Como podemos ver en la Figura 3:5, el funcionamiento es el esperado.

```

Booting from Hard Disk...
cpu0: starting 0
sb: size 200000 nblocks 19937 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap s
tart 58
init: starting sh
$ tsbrk2
.....Desbordamiento de pila
$

```

Figura 3:5

### 3.2.3. Verificación del correcto funcionamiento de fork(), exit() y wait()

El problema de direcciones virtuales sin memoria reservada solamente ocurre con la llamada al sistema *fork* debido a que a la hora de copiar memoria de un proceso padre a un proceso hijo, comprueba la memoria que se ha reservado para el hijo.

Para resolver este problema hay que comprobar que si la memoria que se ha reservado no está disponible, simplemente se ignore en lugar de terminar la ejecución con un panic. Esto se ha resuelto cambiando la llamada a panic por un return 0.

Esta modificación se encuentra en el fichero vm.c en la función *copyuvm()*, Figura 3:6.

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
    pde_t *d;
    pte_t *pte;
    uint pa, i, flags;
    char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyuvm: pte should exist");

        if(!(*pte & PTE_P))
            return 0; //b8-2
    }
}
```

Figura 3:6

Para la comprobación del correcto funcionamiento de la nueva implementación se ha hecho uso del programa de prueba tsbrk4.c proporcionado por los profesores de la asignatura. En este programa se realiza una llamada a la función *fork()*, en ese momento no tenemos memoria reservada, por lo que si se ha realizado una buena modificación del código no saltará ninguna excepción en este test y funcionará correctamente.

Como podemos ver en la Figura 3:7, el funcionamiento es el esperado.

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 20000 nblocks 19937 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap s
tart 58
init: starting sh
$ tsbrk4
Debe imprimir 1: 1.
Debe imprimir 1: 1.
$
```

Figura 3:7

### 3.2.4. Uso correcto del kernel de páginas de usuario sin reservar

En este caso, hay que realizar modificación del código del fichero trap.c, ya que hasta entonces se comprobaba si se había producido un fallo de página en modo kernel y no se debe distinguir si el fallo de página se está dando en modo núcleo o modo usuario para reservar memoria.

```
if(myproc() == 0 || (tf->cs&3) == 0){  
    // In kernel, it must be our mistake.  
    cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",  
           tf->trapno, cpuid(), tf->eip, rcr2());  
    panic("trap");  
}
```

Figura 3:8

Para comprobar que quitando lo seleccionado en la Figura 3:8 este caso funciona correctamente, se ha hecho uso del programa de prueba tsbrk3.c proporcionado por los profesores. Como se puede ver en la Figura 3:9 el funcionamiento es el esperado.

```
Booting from Hard Disk...  
cpu0: starting 0  
sb: size 20000 nblocks 19937 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap s  
tart 58  
init: starting sh  
$ tsbrk3  
Debe imprimir los 50 primeros caracteres de README:  
xv6 is a re-implementation of Dennis Ritchie's and  
$ _
```

Figura 3:9

## 4. Boletín IX

A lo largo del siguiente apartado, se va a realizar la explicación del código desarrollado para la gestión de ficheros de gran tamaño.

### 4.1. Ejercicio I

En este primer ejercicio, se pide la gestión de la creación de los nodos-i que componen la estructura del fichero. Inicialmente, el nodo-i definido en xv6 está compuesto por 12 bloques directos y un bloque simplemente indirecto. En total son  $12+128 = 140$  bloques. Los 128 son los números de bloque que contiene un bloque indirecto.

Para la realización de este ejercicio, primero, se ha añadido la llamada al sistema *big* del mismo modo que se ha llevado a cabo en el boletín VII para poder llevar a cabo la creación de un fichero de gran tamaño mediante la invocación del comando *big*.

Tras esto, se ha procedido a realizar el tratamiento de la gestión de nodos-i. Para ello, se ha necesitado incluir la opción de qemu *QEMUEXTRA* *-= snapshot* a fin de poder crear ficheros de mayor tamaño de lo que permite xv6.

Además de esto, en el fichero *Makefile*, se ha modificado la cantidad de CPUs disponibles a fin de agilizar la creación del fichero de gran tamaño.

Se ha ampliado, como se puede ver en la *Figura 4:0*, el tamaño del sistema de ficheros.

```
#define NBUF      (MAXOPBLOCKS*3) // size of disk block cache
#define FSSIZE    20000 // size of file system in blocks
```

Figura 4:0

En la estructura definida para el tratamiento de los nodos, se ha modificado la cantidad de direcciones de bloque disponibles a fin de incluir los doblemente indirectos. Para poder implementar un bloque doblemente indirecto tenemos que cambiar el número de bloques del nodo-i. Ahora, los números de bloques directos pasan a ser once, para así tener un número o entrada que apunte a un bloque simplemente indirecto y otro número o entrada que apunte al doblemente indirecto. Para ello, se ha de definir en el fichero *fs.h* las siguientes constantes, *Figura 4:1*.

```
#define NDIRECT 11
#define NINDIRECT (BSIZE / sizeof(uint))
#define DOBLEINDIRECT NINDIRECT*NINDIRECT
#define MAXFILE (NDIRECT + NINDIRECT + DOBLEINDIRECT)

// On-disk inode structure
struct dinode {
    short type;           // File type
    short major;          // Major device number (T_DEV only)
    short minor;          // Minor device number (T_DEV only)
    short nlink;          // Number of links to inode in file system
    uint size;             // Size of file (bytes)
    uint addrs[NDIRECT+1+1]; // Data block addresses
};
```

Figura 4:1

Como podemos ver en la figura anterior, dado que se ha modificado la definición de la constante NDIRECT necesitamos cambiar el tamaño de la variable `addrs[]` en la struct `dinode` y también la definición de ésta en la estructura `inode` definida en el fichero `file.h`, ya que `addr[]` debe de ser de tamaño 13, ya que los primeros 11 deben contener los bloques directos, el duodécimo un único bloque indirecto y el decimotercero debe ser el bloque doblemente indirecto.

Como hemos cambiado la definición de la constante NDIRECT tenemos que crear una nueva imagen del sistema de ficheros, creada en el fichero `fs.img`, ya que `mkfs` usa NDIRECT para construir los sistemas de ficheros iniciales. Por lo tanto, borrando `fs.img` bastará, porque al ejecutar de nuevo la instrucción `'make'` se creará una nueva imagen.

En el fichero `fs.c`, se encuentra la implementación de `bmap` que se trata de la función encargada de llevar a cabo la asignación de los bloques de datos a la estructura. La función `bmap` recibe dos parámetros: `*ip`, que es un puntero al nodo-`i`, y `bn` de tipo `uint`, que es el número de bloque lógico.

En primer lugar, se comprueba que el número de bloque lógico esté entre los bloques directos. Si lo está, pero el bloque no está creado, es decir, si `ip->addr[bn]` es 0 entonces tenemos que crearlo con `balloc` pasándole el número de dispositivo donde está alojado el fichero. Si se crea, `balloc` devuelve la dirección del bloque creado, que se guarda también en `ip->addr[bn]`. Si no se crea, entonces se devuelve la dirección del bloque directo.

Si el bloque lógico no está entre los directos tenemos que consultar en el bloque simplemente indirecto. Para ello lo que tenemos que hacer es restar al bloque lógico el número de entradas directas para situarnos en el bloque simplemente indirecto. Ahora, comprobamos si el bloque lógico está en el simplemente indirecto. Si lo está, tenemos que comprobar si este existe. Si no existe, se crea con `balloc` y se guarda en `ip->addr[NDIRECT]` la dirección que devuelve. A continuación, guardamos en un buffer llamado `bp` el contenido del bloque cuya dirección se pasa como parámetro que en este caso es la del bloque simplemente indirecto.

El siguiente paso es obtener la dirección del bloque directo, para ello alojamos en la variable `a` la dirección del array que contiene los bloques directos a partir del buffer que es `bp` (`bp->data`). Se comprueba si el bloque lógico pertenece a ese array. Si no pertenece entonces se crea el bloque directo con `balloc` y se llama a `log_write`. Esta función escribe en disco y en la bitácora del disco el buffer `bp` pasado como parámetro.

Escribe en la bitácora porque si se produce algún fallo en el sistema, escribir en disco el buffer cuando se inicie de nuevo. Tanto si se crea como si no, se llama a `brelse()`. Por último, se devuelve la dirección del bloque. Si el bloque lógico no está en el bloque simplemente indirecto, tenemos que consultar el bloque doblemente indirecto. Para ello, se ha incluido el código observado en la *Figura 4:2*.

```

// Como no pertenece al bloque simplemente indirecto, se resta su numero
// de entradas (128) para poder situarnos en el doblemente indirecto
bn -= NINDIRECT;

if(bn < DOBLEINDIRECTO){

    // Cargar un bloque doblemente indirecto alojandolo si es necesario.
    if((addr = ip->addrs[NDIRECT+1]) == 0)
        ip->addrs[NDIRECT+1] = addr = balloc(ip->dev);
    bp = bread(ip->dev, addr);
    a = (uint*)bp->data;

    // Crea el bloque simplemente indirecto si este no esta creado
    if((addr = a[bn/NINDIRECT]) == 0){
        a[bn/NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp);
    }

    // Se carga el bloque simplemente indirecto
    struct buf *bp2;
    bp2 = bread(ip->dev, addr);
    a = (uint*)bp2->data;

    // Se crea el bloque directo si no está creado
    // Con el % accedemos a la entrada correspondiente del bn en el bloque directo
    if((addr = a[bn%NINDIRECT]) == 0){
        a[bn%NINDIRECT] = addr = balloc(ip->dev);
        log_write(bp2);
    }
    // Se carga el bloque directo si ya está creado
    else
        addr = a[bn%NINDIRECT];

    brelse(bp2);
    brelse(bp);
    return addr;
}
}

```

Figura 4:2

Primero tenemos que situarnos en el bloque doblemente indirecto, para ello tenemos que restar al bloque lógico el número de entradas del bloque simplemente indirecto que son 128. A continuación comprobamos si el bloque doblemente indirecto está creado o no. Si no existe, se crea con `balloc` alojando la dirección en `ip->addrs[NDIRECT+1]`.

Como hemos comentado anteriormente, guardamos en un buffer llamado `bp` el contenido del bloque cuya dirección se pasa como parámetro que este caso es la del bloque doblemente indirecto. La variable `a`, cuyo valor es la dirección del array que contiene los bloques simplemente indirectos a partir del buffer que es `bp` (`bp->data`). Ahora, comprobamos si el bloque simplemente indirecto asociado a la posición `bn/NINDIRECT` del array del bloque doblemente indirecto existe o no.

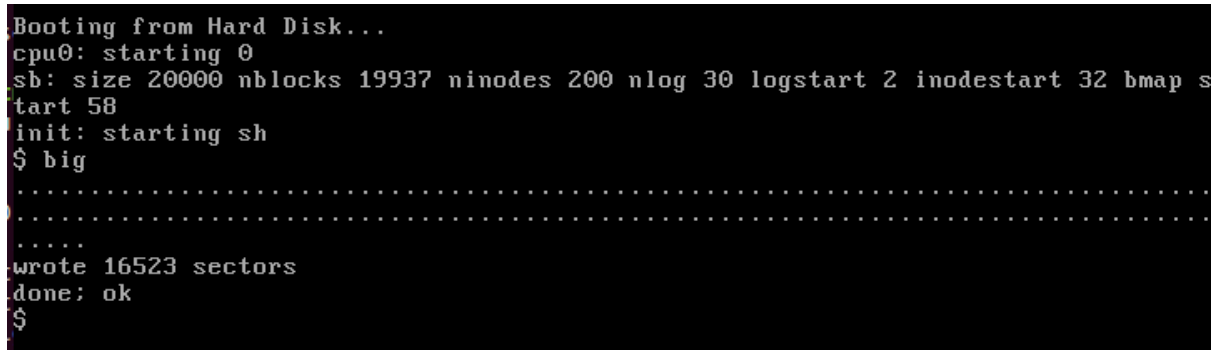
Si no existe se crea con `balloc` y se hace la llamada a `log_write`. Cuando se crea se guarda en `a[bn/NINDIRECT]`, la dirección que devuelve `balloc`.

Tanto si se crea como si ya está creado, el bloque simplemente indirecto correspondiente a `a[bn/NINDIRECT]` se tiene que leer para hallar el bloque directo. Para ello, alojamos en la variable `bp2` de tipo `struct buf` el contenido del bloque simplemente indirecto cuya dirección se pasa como parámetro. Ahora, modificamos la variable `a` con la dirección del array de bloques directos que contiene el simplemente indirecto mencionado. Por último, comprobamos si el bloque directo existe.



Para obtener el bloque directo calculamos el módulo del bloque lógico respecto al número de entradas simplemente indirecto. Si el módulo es 0 quiere decir que el bloque directo no existe y se tiene que crear. Para crearlo se usa `balloc` y se guarda en `a[bn%NINDIRECT]` la dirección que devuelve, luego se llama a `log_write`. Si es distinto de 0 entonces existe y simplemente se devuelve su dirección. Para terminar, se llama a `brelse()` dos veces, una con `bp` y la otra con `bp2`.

Una vez terminada la implementación, realizamos la prueba de que esta implementación funciona correctamente utilizando el test `big.c` proporcionado por los profesores de la asignatura. Como podemos ver en la Figura 4:3, el funcionamiento es correcto.



```
Booting from Hard Disk...
cpu0: starting 0
sb: size 20000 nblocks 19937 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap s
start 58
init: starting sh
$ big
.....
.....
wrote 16523 sectors
done: ok
$
```

Figura 4:3

## 4.2. Ejercicio II

El segundo ejercicio propuesto, consiste en la modificación de la eliminación de los bloques reservados en la creación de ficheros. Para ello, se deberá de modificar la función `itrunc` perteneciente al fichero `fs.c`.

En la Figura 4:4, podemos observar el código implementado para llevar a cabo la eliminación de los nodos doblemente indexados en las reservas.

Si se detecta la existencia de un bloque doblemente indexado, se procederá a comprobar las direcciones de memoria de los distintos bloques simplemente indexados que lo componen y, finalmente, comprobará aquellos que apunten a un bloque de datos.

Durante este proceso, se liberarán aquellos bloques que estén siendo apuntados por la estructura de memoria y, dentro de la propia estructura de nodos-i, se cambiarán los valores de las direcciones de memoria por un valor cero.

De este modo, todos los bloques que componen el fichero serán liberados.

```

//Detecta que existe un BDI
if(ip->addrs[NDIRECT+1] )
{
    //Lee el BDI
    bp = bread(ip->dev, ip->addrs[NDIRECT+1]);
    a = (uint *)bp->data;
    //Para cada entrada del BDI se deberán de eliminar todos los BSI y los bloques de datos asociados a estos
    for(i = 0; i < NINDIRECT; i++)
    {
        //Compruebo si se dispone de un BSI con dirección de bloque de disco, consiste en la modificación de la
        //Si contiene un BSI, se recorre y se liberan los bloques de datos
        if(a[i])
        {
            bp2 = bread(ip->dev, a[i]);
            b = (uint*)bp2->data;
            //Libero los bloques de datos del BSI
            for (j = 0; j < NINDIRECT; j++)
            {
                if(b[j])
                    bfree(ip->dev, b[j]);
            }
            brelse(bp2);
            //Libero el BSI
            bfree(ip->dev, a[i]);
        }
    }
    ip->addrs[NDIRECT+1] = 0;
}

```

Figura 4:4

En la *Figura 4:5*, se muestran los resultados de las pruebas de creación y eliminación del fichero generado por el comando `big`. Como podemos ver, al realizar la eliminación del fichero `big.file`, si el borrado que se ha implementado no fuera correcto saltaría una excepción para notificarnos que no ha sido posible crear el fichero, dado que esto no es lo que sucede, podemos afirmar que nuestra implementación es correcta.

```

init: starting sh
$ big
.....
wrote 16523 sectors
done: ok
$ rm big.file
$ big
.....
wrote 16523 sectors
done: ok

```

Figura 4:5