Boletín 8: Ficheros grandes en xv6

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2021/2022

Introducción

- xv6 trata todos los dispositivos sin la ayuda de la BIOS
- Por lo tanto, implementa el acceso a los dispositivos a bajo nivel
- No tiene el concepto de *driver*, aunque veremos lo que se puede considerar un *driver*

Tratamiento de interrupciones

- Tanto las interrupciones *hardware* como las *software* se tratan en un único punto: la función `trap()`, que recibe como parámetro un `struct trapframe`:

```
1  void
   trap(struct trapframe *tf)
3  {
     if(tf->trapno == T_SYSCALL){
5      if(proc->killed)
         exit();
7      proc->tf = tf;
       syscall();
9      if(proc->killed)
         exit();
11     return;
     }
13
     switch(tf->trapno){
15   case T_IRQ0 + IRQ_TIMER:
       if(cpunum() == 0){
17       acquire(&tickslock);
         ticks++;
19       wakeup(&ticks);
         release(&tickslock);
21     }
       lapiceoi();
23     break;
```

Tratamiento de interrupciones (cont.)

```
      case T_IRQ0 + IRQ_IDE:
25      ideintr();
        lapiceoi();
27      break;
    // ...
```

- Como ya se vio, todas las llamadas al sistema van a la función `syscall()`
- Pero las interrupciones del *hardware* entran en el `switch`
- Todas las interrupciones *hardware* se tienen que terminar con un EOI (*end of interrupt*) (función `lapiceoi()`)
- xv6 establece el temporizador para que utilice la IRQ 0, y para que produzca la interrupción 32 (se realiza en `idtinit()`)

Driver de dispositivo: disco

- En general, el *driver* es el código del SO que trata con el dispositivo a bajo nivel, inicia las transacciones de E/S y recoge los datos
- xv6 no tiene una línea marcada exactamente de qué es un *driver*, pero sí que intenta mantener en un mismo fichero las funciones de bajo nivel para tratar con un dispositivo en particular
- Por ejemplo, el *driver* de disco inicia las transferencias y recoge los resultados (en `ide.c`)
- Para recoger los resultados es llamado desde `trap()`
- El *hardware* abstrae el disco como un conjunto de bloques desde el 0 al tamaño del disco, con un tamaño que suele ser de 512 bytes
- El SO por su parte gestiona los bloques de disco con una estructura `struct buf`

Driver de dispositivo: disco (cont.)

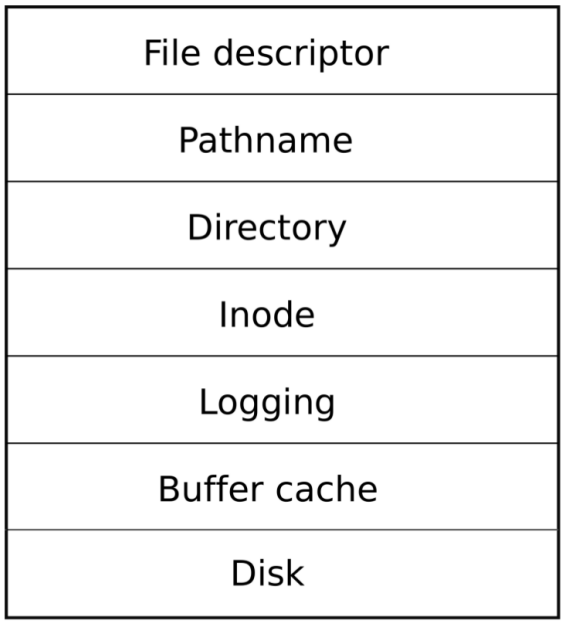
```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct sleeplock lock;
  uint refcnt;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- `BSIZE` corresponde al tamaño del bloque de disco, 512 bytes
- `dev` y `blockno`: dispositivo y bloque que se lee o escribe
- `B_VALID` especifica que el bloque se leyó
- `B_DIRTY` especifica que el bloque se debe escribir a disco
- `B_BUSY` bit de cerrojo. Indica que un proceso está usando el buffer. Cuando un buffer tiene este bit a uno se dice que el buffer está bloqueado

El sistema de ficheros de xv6

- El sistema de ficheros de xv6 proporciona ficheros, directorios y rutas al estilo Unix
- La implementación se organiza en siete capas tal y como se muestra en la figura:

- La capa de disco lee y escribe bloques en un disco duro IDE
- La capa de buffer de caché almacena los bloques de disco y sincroniza el acceso a ellos
- La capa de bitácora (*logging*) permite a las capas superiores agrupar varias actualizaciones de bloques en una transacción
- La capa de nodo-i provee los ficheros individuales, cada uno representado a través de un número-i único y algunos bloques conteniendo los datos

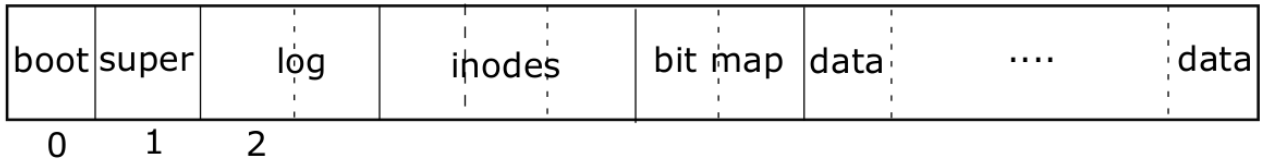


El sistema de ficheros de xv6 (cont.)

- La capa de directorio implementa cada directorio como un tipo especial de fichero cuyo contenido es una secuencia de entradas de directorio (pares de nombre de fichero y número-i)
- La capa de *pathname* proporciona rutas jerárquicas a ficheros como `/usr/rtm/xv6/fs.c` y las resuelve mediante búsqueda recursiva
- La capa de descriptores de fichero abstrae los recursos Unix (p.e. tuberías, dispositivos, ficheros, etc.) usando el interfaz del sistema de ficheros

El sistema de ficheros de xv6 (cont.)

- xv6 divide el disco en varias secciones tal y como se muestra en la figura:



- El sistema de ficheros no usa el bloque 0 que contiene el sector de arranque
- El bloque 1 se denomina *superbloque* y contiene los metadatos del sistema de ficheros (que se inicializan con el programa `mkfs` al construir el sistema de ficheros)
  - Tamaño del sistema de ficheros en bloques, número de bloques de datos, número de nodos-i y número de bloques en el *log*
- Los bloques a partir del 2 contienen el *log*
- A continuación los nodos-i (con múltiples nodos-i por bloque)
- Después tenemos los bloques de *bitmaps* que permiten saber los bloques de datos en uso
- El resto son bloques de datos

La caché de buffers (`bio.c`)

- La caché de buffers del sistema de ficheros en xv6 realiza dos tareas:
  - Sincronizar el acceso a los bloques de disco, asegurándose de que:
    - sólo exista una copia de un bloque en memoria
    - sólo un hilo kernel usa esa copia en un instante determinado
  - Guardar los bloques más usados de tal manera que no tenga que volverse a leer del disco
- La interfaz exportada por la caché de *buffers* consiste en las operaciones `bread()` y `bwrite()`
  - `bread()` devuelve un *buffer* con una copia de un bloque que puede leerse y modificarse en memoria
  - `bwrite()` escribe un *buffer* modificado al bloque apropiado del disco
- La sincronización de los accesos se consigue permitiendo que como máximo haya un hilo referenciando a un bloque del *buffer*
  - Si otro hilo llama a `bread()` sobre el mismo bloque se quedará esperando hasta que el bloque se libere mediante `brelse()`

La capa de bitácora (`log.c`)

- xv6 implementa el concepto de bitácora para resolver el problema de una caída del sistema durante las operaciones al sistema de ficheros que pueda dejar al sistema de archivos en un estado corrupto
- Las llamadas al sistema en xv6 no escriben directamente en el disco
  - En su lugar, introducen una descripción de todas las escrituras a disco que desean hacer en la bitácora del disco
  - Después escriben un registro de *commit* para indicar que la bitácora contiene una operación completa
  - En ese momento la llamada al sistema copia las escrituras en el sistema de ficheros y cuando finalizan borra las entradas de la bitácora del disco

La capa de bitácora (`log.c`) (cont.)

- Un uso típico de la bitácora en una llamada al sistema se parece a:

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

- Como se observa, el código de la llamada al sistema indica el comienzo y el fin de una secuencia de escrituras que deben ser atómicas
- Por eficiencia, y para permitir cierta concurrencia, el sistema de bitácora puede acumular las escrituras de múltiples llamadas al sistema en una transacción
- Para evitar interrumpir una llamada al sistema por una transacción, el sistema de bitácora sólo realiza el *commit* cuando no hay llamadas al sistema en ejecución

La capa de bitácora (`log.c`) (cont.)

- Aunque el driver IDE del xv6 no permite trabajar por lotes, el diseño del sistema de ficheros permite realizar el *commit* de varias transacciones y convertirlas en una única operación al driver del disco

La capa de nodos-i (`fs.c`)

- El término nodo-i tiene dos significados relacionados:
  - Se puede referir a la estructura de datos del disco que contiene los metadatos y la lista de bloques de datos de un fichero
  - También se puede referir al nodo-i en memoria que contiene una copia del nodo-i del disco, así como información extra que necesita el *kernel*
- Todos los nodos-i del disco están empaquetados en un área contigua del disco denominada bloques de nodos-i
- Todos los nodos-i tienen el mismo tamaño
  - Dado un número *n* es sencillo identificar el n-ésimo nodo-i del disco. A este número se le denomina número-i o número del nodo-i

La capa de nodos-i (`fs.c`) (cont.)

- La estructura de un nodo-i en el disco está definida en **`struct dinode`**:

```
#define NDIRECT 12

// On-disk inode structure
struct dinode {
  short type;           // File type: files, directories, and special files
  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];   // Data block addresses
};
```

- Por otro lado, el kernel mantiene en memoria el conjunto de nodos-i activos (con punteros C referenciando a ese nodo-i)

La capa de nodos-i (`fs.c`) (cont.)

- `struct inode` es la copia en memoria de una `struct dinode` en disco:

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock;
  int flags;          // I_VALID

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

- El campo `ref` mantiene el número de punteros C que referencia al nodo-i en memoria
- El kernel descarta el nodo-i cuando dicha cuenta llega a cero
- Las funciones `iget()` e `iput()` adquieren y liberan punteros a un nodo-i, modificando el valor de `ref`

La capa de directorio

- Un directorio se implementa internamente de forma muy similar a un fichero
- El campo de tipo de su nodo-i es `T_DIR` y sus datos son una secuencia de entradas de directorio. Cada entrada es una `struct dirent`:

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

- Si el nombre tiene menos de 14 caracteres se termina con un byte `NUL` (0)
- Las entradas con `inum` cero están libres
- La función `dirlookup()` busca en un directorio una entrada con un nombre dado
  - Si la encuentra devuelve un puntero, no bloqueado, al nodo-i y establece `*poff` al *offset* de la entrada dentro del directorio (por si desea editarla)

La capa de directorio (cont.)

```
// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
struct inode* dirlookup(struct inode *dp, char *name, uint *poff)
```

- La función **`dirlink()`** escribe una nueva entrada de directorio con el nombre dado y el número-i en el directorio indicado.

```
// Write a new directory entry (name, inum) into the directory dp.
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  struct inode *ip;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }
  ...
```

Las rutas de nombres

- La búsqueda de un fichero en una ruta implica una sucesión de llamadas a `dirlookup()`, una por cada componente de la ruta
- `namei()` evalúa el camino y devuelve el correspondiente nodo-i
- El trabajo se realiza en la función `namex()`
- `namex()` comienza decidiendo si la evaluación debe realizarse desde el directorio raíz o desde el directorio actual
- A continuación usa `skipelem()` para considerar cada elemento de la ruta
  - En cada iteración se debe buscar en nombre en el nodo-i actual
  - La iteración comienza bloqueando el nodo-i y comprobando que se trata de un directorio
  - Cuando se encuentra una coincidencia se devuelne el nodo-i correspondiente

La capa de descriptores de fichero

- Uno de los mejores aspectos de la interfaz de Unix es que la mayoría de recursos se representan como un fichero
- La capa de descriptores de fichero es la responsable de conseguir esta uniformidad
- xv6 proporciona a cada proceso su propia tabla de ficheros abiertos (o descriptores de ficheros)
- Cada fichero abierto se representa por una `struct file` que es una envoltura sobre un nodo-i o una tubería, junto con un puntero de E/S
- Cada llamada a `open()` crea un nuevo fichero abierto
  - Si varios procesos abren el mismo fichero de forma independiente, las diferentes instancias tendrán diferentes punteros de E/S
  - Por otro lado, un mismo fichero abierto puede aparecer múltiples veces en la tabla de un proceso (y de múltiples procesos)
  - Un contador de referencias controla el número de referencias existentes a un fichero abierto

La capa de descriptores de fichero (cont.)

- También se definen funciones que permiten reservar un fichero (`filealloc()`), crear una referencia duplicada (`filedup()`), liberar una referencia (`fileclose()`) y leer y escribir datos (`fileread()` y `filewrite()`)

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```

- La estructura `proc` guarda el conjunto de ficheros abiertos de cada proceso en su campo `ofile`
- La posición i-ésima de ese array `ofile` corresponde al *handle* de fichero abierto en el proceso (`ofile[0]` corresponde al fichero 0, etc.)

La capa de descriptores de fichero (cont.)

```
1  // Per-process state
   struct proc {
3    uint sz;                     // Size of process memory (bytes)
     pde_t* pgdir;                // Page table
5    char *kstack;                // Bottom of kernel stack for this process
     enum procstate state;        // Process state
7    int pid;                     // Process ID
     struct proc *parent;         // Parent process
9    struct trapframe *tf;        // Trap frame for current syscall
     struct context *context;     // swtch() here to run process
11   void *chan;                  // If non-zero, sleeping on chan
     int killed;                  // If non-zero, have been killed
13   struct file *ofile[NOFILE];  // Open files
     struct inode *cwd;           // Current directory
15   char name[16];               // Process name (debugging)
   };
```

- La tabla global `ftable` almacena todos los ficheros abiertos en el sistema

Ficheros grandes en xv6

- En este boletín incrementaremos el tamaño máximo de un fichero en xv6
- Actualmente, el tamaño de los ficheros en xv6 está limitado a 140 sectores (71.680 bytes)
- Este límite viene del hecho de un nodo-i en xv6 contiene 12 números de bloque directos y un único número de bloque indirecto para un total de $12 + 128 = 140$ bloques
  - Un bloque indirecto contiene hasta 128 números de bloque adicionales
- Para ello cambiaremos el código del sistema de fichero de xv6 para soportar un bloque doblemente indirecto en cada nodo-i
  - Que contendrá 128 direcciones de bloques simplemente indirectos
- Como resultado, un fichero podrá estar constituido de hasta 16.523 sectores (o cerca de 8,5 megabytes)

Preliminares

- Añade `QEMUEXTRA=-snapshot` justo antes de `QEMUOPTS` en el `Makefile`. Este paso aumenta la velocidad de qemu cuando xv6 crea ficheros grandes
- `mkfs` inicializa el sistema de fichero con menos de 1000 bloques libres, demasiados pocos para los cambios que queremos realizar. Modifica `param.h` para establcer el valor de `FSSIZE` a:

```
#define FSSIZE       20000  // size of file system in blocks
```

- Baja el fichero `big.c` al directorio xv6 y añadelo a la lista `UPROGS`. Arranca xv6 y ejecuta `big`
  - Este programa crea un fichero tan grande como le permita xv6, indicando el tamaño resultante
  - Debería decir 140 sectores

Dónde mirar

- El formato de un nodo-i en el disco está definido en `fs.h`. Nos interesa mirar los elementos `NDIRECT`, `NINDIRECT`, `MAXFILE` y `addrs[]` de la estructura `dinode`
- El código que encuentra los datos de un fichero en disco está en `bmap()` (en `fs.c`). Estudia dicho código hasta entenderlo
  - `bmap()` se llama tanto cuando se **lee** como cuando se **escribe** en un fichero
  - Cuando se escribe, `bmap()` reserva tantos bloques como sean necesarios para guardar el contenido del fichero. También reserva un bloque indirecto si lo necesita para guardar las direcciones del bloque
- `bmap()` trabaja dos tipos de números de bloque
  - El argumento `bn` es un *bloque lógico* (un número de bloque relativo al comienzo del fichero)
  - Los números de bloque en `ip->addrs[]` y el argumento que se le pasa a `bread()`, son números de bloque de disco. `bmap()` mapea los bloques lógicos de un fichero a números de bloque en disco

Implementación a realizar

- **EJERCICIO 1**: Modifica `bmap()` para que implemente un bloque doblemente indirecto. No puedes cambiar el tamaño del nodo-i en disco, por lo que sacrificaremos un bloque directo para implementar el doblemente indirecto
- Los primeros 11 elementos de `ip->addrs[]` deben contener bloques directos, el duodécimo un único bloque indirecto y el decimotercero debe ser tu bloque doblemente indirecto
- Si todo va bien, el programa `big` informará que ha podido escribir 16523 sectores. Necesitará bastante tiempo para terminar la ejecución
- **EJERCICIO 2**: Modifica xv6 para manejar el borrado de ficheros con bloques doblemente indirectos

Cosas a tener en cuenta

- Si cambias la definición de `NDIRECT` necesitarás cambiar el tamaño de `addrs[]` en la `struct inode` (en `file.h`)
- Asegúrate que `struct inode` y `struct dinode` tienen el mismo número de elementos en sus arrays `addrs[]`
- Si cambias la definición de `NDIRECT` asegúrate de crear una nueva imagen del sistema de ficheros (`fs.img`), ya que `mkfs` usa `NDIRECT` para construir los sistemas de ficheros iniciales.
  - Si borras `fs.img`, `make` creará una nueva imagen
- Si tu sistema de ficheros se corrompe, borra `fs.img` para crear una nueva imagen limpia
- No olvides hacer `brelse()` para cada bloque sobre el que hagas `bread()`
- Sólo debes reservar los bloques indirectos y los doblemente indirectos cuando sea necesario, al igual que ocurre en el `bmap()` original