Boletín 9: Ficheros grandes en xv6

Ampliación de Sistemas Operativos

Dpto. Ingeniería y Tecnología de Computadores (DITEC)

Universidad de Murcia

Curso 2017/2018

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**: 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
- **OPCIONAL**: Modifica xv6 para manejar el borrado de ficheros con bloques doblemente indirectos (Nota: para obtener la puntuación máxima en este boletín no es necesario realizar esta parte opcional)

Cosas a tener en cuenta

- Si cambias la definición de `NDIRECT` necesitaras 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 fichero 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