Sistemas operativos II

Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

Punto de vista del usuario

- estructura jerárquica
- posibilidad de crear y borrar ficheros
- crecimiento dinámico de los ficheros
- protección de los datos de los ficheros
- tratamiento de los dispositivos periféricos como ficheros
- cada fichero tiene un nombre completo que es una secuencia de nombres separados por el caracter /; cada uno de los nombres designa un nombre único en el nombre previo
- un fichero es una sucesión de bytes
- un directorio es un fichero normal
- permisos de acceso controlados (lectura, escritura, ejecución) rwxrwxrwx
- árbol con un nodo raiz (/):
  - cada nodo que no es una hoja es un directorio
  - cada hoja: fichero, directorio o dispositivo
- en realidad se trata de un grafo y no de un árbol: enlaces reales y simbólicos

Punto de vista del kernel

- un fichero es una sucesión de bytes
- un fichero está representado por una estructura pequeña, con la informacion que el kernel necesita conocer de dicho fichero, denominada inodo
  - propietario y grupo del fichero (*uid y gid*)
  - *modo* del fichero: entero codificado bit a bit con los permisos y el tipo de fichero
  - fechas (último acceso, última modificación, último cambio en el inodo)
  - tamaño
  - número de enlaces reales
  - direcciones de disco que ocupa

inodos en openBSD

```
struct  ufs1_dinode {
    u_int16_t       di_mode;        /*   0: IFMT, permissions; see below. */
    int16_t         di_nlink;       /*   2: File link count. */
    union {
            u_int16_t oldids[2];    /*   4: Ffs: old user and group ids. */
            u_int32_t inumber;      /*   4: Lfs: inode number. */
    } di_u;
    u_int64_t       di_size;        /*   8: File byte count. */
    int32_t         di_atime;       /*  16: Last access time. */
    int32_t         di_atimensec;   /*  20: Last access time. */
    int32_t         di_mtime;       /*  24: Last modified time. */
    int32_t         di_mtimensec;   /*  28: Last modified time. */
    int32_t         di_ctime;       /*  32: Last inode change time. */
    int32_t         di_ctimensec;   /*  36: Last inode change time. */
    ufs1_daddr_t    di_db[NDADDR];  /*  40: Direct disk blocks. */
    ufs1_daddr_t    di_ib[NIADDR];  /*  88: Indirect disk blocks. */
    u_int32_t       di_flags;       /* 100: Status flags (chflags). */
    int32_t         di_blocks;      /* 104: Blocks actually held. */
    int32_t         di_gen;         /* 108: Generation number. */
    u_int32_t       di_uid;         /* 112: File owner. */
    u_int32_t       di_gid;         /* 116: File group. */
    int32_t         di_spare[2];    /* 120: Reserved; currently unused */
};
```

inodos en openBSD

```
struct ufs2_dinode {
    u_int16_t       di_mode;        /*   0: IFMT, permissions; see below. */
    int16_t         di_nlink;       /*   2: File link count. */
    u_int32_t       di_uid;         /*   4: File owner. */
    u_int32_t       di_gid;         /*   8: File group. */
    u_int32_t       di_blksize;     /*  12: Inode blocksize. */
    u_int64_t       di_size;        /*  16: File byte count. */
    u_int64_t       di_blocks;      /*  24: Bytes actually held. */
    ufs_time_t      di_atime;       /*  32: Last access time. */
    ufs_time_t      di_mtime;       /*  40: Last modified time. */
    ufs_time_t      di_ctime;       /*  48: Last inode change time. */
    ufs_time_t      di_birthtime;   /*  56: Inode creation time. */
    int32_t         di_mtimensec;   /*  64: Last modified time. */
    int32_t         di_atimensec;   /*  68: Last access time. */
    int32_t         di_ctimensec;   /*  72: Last inode change time. */
    int32_t         di_birthnsec;   /*  76: Inode creation time. */
    int32_t         di_gen;         /*  80: Generation number. */
    u_int32_t       di_kernflags;   /*  84: Kernel flags. */
    u_int32_t       di_flags;       /*  88: Status flags (chflags). */
    int32_t         di_extsize;     /*  92: External attributes block. */
    ufs2_daddr_t    di_extb[NXADDR];/*  96: External attributes block. */
    ufs2_daddr_t    di_db[NDADDR];  /* 112: Direct disk blocks. */
    ufs2_daddr_t    di_ib[NIADDR];  /* 208: Indirect disk blocks. */
    int64_t         di_spare[3];    /* 232: Reserved; currently unused */
};
```

inodo en ext2

```
struct ext2fs_dinode {
    u_int16_t       e2di_mode;      /*   0: IFMT, permissions; see below. */
    u_int16_t       e2di_uid_low;   /*   2: Owner UID, lowest bits */
    u_int32_t       e2di_size;      /*      4: Size (in bytes) */
    u_int32_t       e2di_atime;     /*      8: Access time */
    u_int32_t       e2di_ctime;     /*     12: Create time */
    u_int32_t       e2di_mtime;     /*     16: Modification time */
    u_int32_t       e2di_dtime;     /*     20: Deletion time */
    u_int16_t       e2di_gid_low;   /*  24: Owner GID, lowest bits */
    u_int16_t       e2di_nlink;     /*  26: File link count */
    u_int32_t       e2di_nblock;    /*  28: Blocks count */
    u_int32_t       e2di_flags;     /*  32: Status flags (chflags) */
    u_int32_t       e2di_linux_reserved1; /* 36 */
    u_int32_t       e2di_blocks[NDADDR+NIADDR]; /* 40: disk blocks */
    u_int32_t       e2di_gen;       /* 100: generation number */
    u_int32_t       e2di_facl;      /* 104: file ACL (not implemented) */
    u_int32_t       e2di_dacl;      /* 108: dir ACL (not implemented) */
    u_int32_t       e2di_faddr;     /* 112: fragment address */
    u_int8_t        e2di_nfrag;     /* 116: fragment number */
    u_int8_t        e2di_fsize;     /* 117: fragment size */
    u_int16_t       e2di_linux_reserved2; /* 118 */
    u_int16_t       e2di_uid_high;  /* 120: 16 highest bits of uid */
    u_int16_t       e2di_gid_high;  /* 122: 16 highest bits of gid */
    u_int32_t       e2di_linux_reserved3; /* 124 */
};
```

Punto de vista del kernel

- un directorio es un fichero normal. Sus contenidos son las *entradas de directorio*, cada una de ellas contiene información de uno de los ficheros en dicho directorio. (basicamente el nombre y el número de inodo)
- cada fichero un único *inodo* pero varios nombres (enlaces)
  - enlace real a un fichero: entrada de directorio que se refiere al mismo inodo
  - enlace simbólico a un fichero: fichero especial que contiene el path al cual es el enlace
- varios tipos de fichero
  - fichero normal
  - directorio
  - dispositivo (bloque o carácter)
  - enlace simbólico
  - fifo
  - socket

constantes en ufs/ufs/dinode.h

```
#define NDADDR  12                      /* Direct addresses in inode. */
#define NIADDR  3                       /* Indirect addresses in inode. */

#define MAXSYMLINKLEN_UFS1      ((NDADDR + NIADDR) * sizeof(ufs1_daddr_t))
#define MAXSYMLINKLEN_UFS2      ((NDADDR + NIADDR) * sizeof(ufs2_daddr_t))

/* File permissions. */
#define IEXEC           0000100         /* Executable. */
#define IWRITE          0000200         /* Writeable. */
#define IREAD           0000400         /* Readable. */
#define ISVTX           0001000         /* Sticky bit. */
#define ISGID           0002000         /* Set-gid. */
#define ISUID           0004000         /* Set-uid. */

/* File types. */
#define IFMT            0170000         /* Mask of file type. */
#define IFIFO           0010000         /* Named pipe (fifo). */
#define IFCHR           0020000         /* Character device. */
#define IFDIR           0040000         /* Directory file. */
#define IFBLK           0060000         /* Block device. */
#define IFREG           0100000         /* Regular file. */
#define IFLNK           0120000         /* Symbolic link. */
#define IFSOCK          0140000         /* UNIX domain socket. */
#define IFWHT           0160000         /* Whiteout. */

#endif /* _UFS_DINODE_H_ */
```

- una instalación puede tener una o varias unidades físicas
- cada unidad física puede tener uno o varios sistemas de ficheros (o unidades lógicas)
- cada sistema de ficheros: sucesión de bloques (grupos de sectores) de 512, 1024, 2048 ...bytes. En Unix System V R2 tiene la siguiente estructuta física

BOOT	SUPER BLOQUE	LISTA INODOS	AREA DE DATOS

- los distintos sistemas de ficheros se *montan* (llamada al sistema *mount*) sobre directorios dando lugar a un único árbol (grafo) de directorios en el sistema.

- el kernel trata sólo con dispositivos lógicos.
  - Cada fichero en el sistema queda perfectamente definido por un número de dispositivo lógico (sistema de ficheros) y número de inodo dentro de ese sistema de ficheros
  - cada bloque queda perfectamente definido por un número de dispositivo lógico (sistema de ficheros) y número de bloque dentro de ese sistema de ficheros
  - las estrategias de asignación y contabilidad se hacen en base a bloques lógicos
- la traducción de direcciones lógicas a direcciones físicas la hace el manejador de dispositivo (*device driver*)

- El uso de ficheros en el sistema está gobernado por tres tablas
  - **tabla de inodos en memoria(inode table)** Contiene los inodos en memoria de los ficheros que están en uso, junto con, entre otras cosas, un contador de referencias. Global del sistema, en el espacio de datos del kernel
  - **tabla ficheros abiertos(file table)** Una entrada por cada apertura de un fichero (varias aperturas del mismo fichero dan lugar a varias entradas). Contiene el modo de apertura (O_RDONLY, O_WRONLY . . .), *offset* en el fichero, contador de referencias y puntero al inodo en la tabla de inodos en memoria. Global del sistema, en el espacio de datos del kernel
  - **tabla de descriptores de fichero de usuario (user file descriptor table)** Cada apertura, o cada llamada *dup()* crean una entrada en esta tabla (asi como la llamada *fork()*). Contiene un referencia a la entrada correspondiente de la tabla ficheros abiertos. Una para cada proceso, en su *u_area*

ejemplo de tablas de ficheros

Si tenemos dos procesos

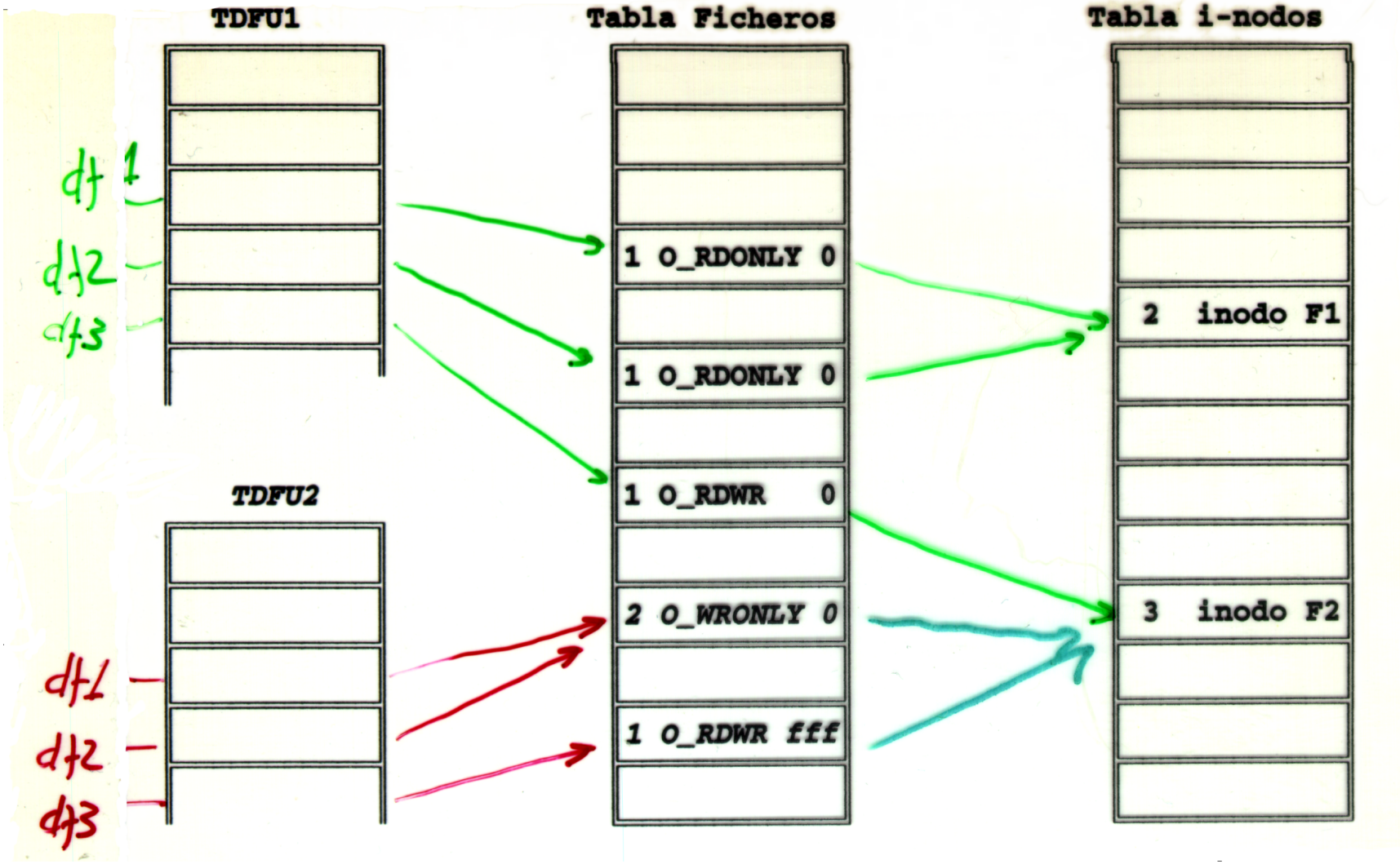
- P1

```
...
df1=open("F1",O_RDONLY);
df2=open("F1",O_RDONLY);
df3=open("F2",O_RDWR);
..
```

- P2

```
...
df1=open("F2",O_WRONLY);
df2=dup(df1);
df3=open("F2",O_RDWR|O_APPEND);
..
```

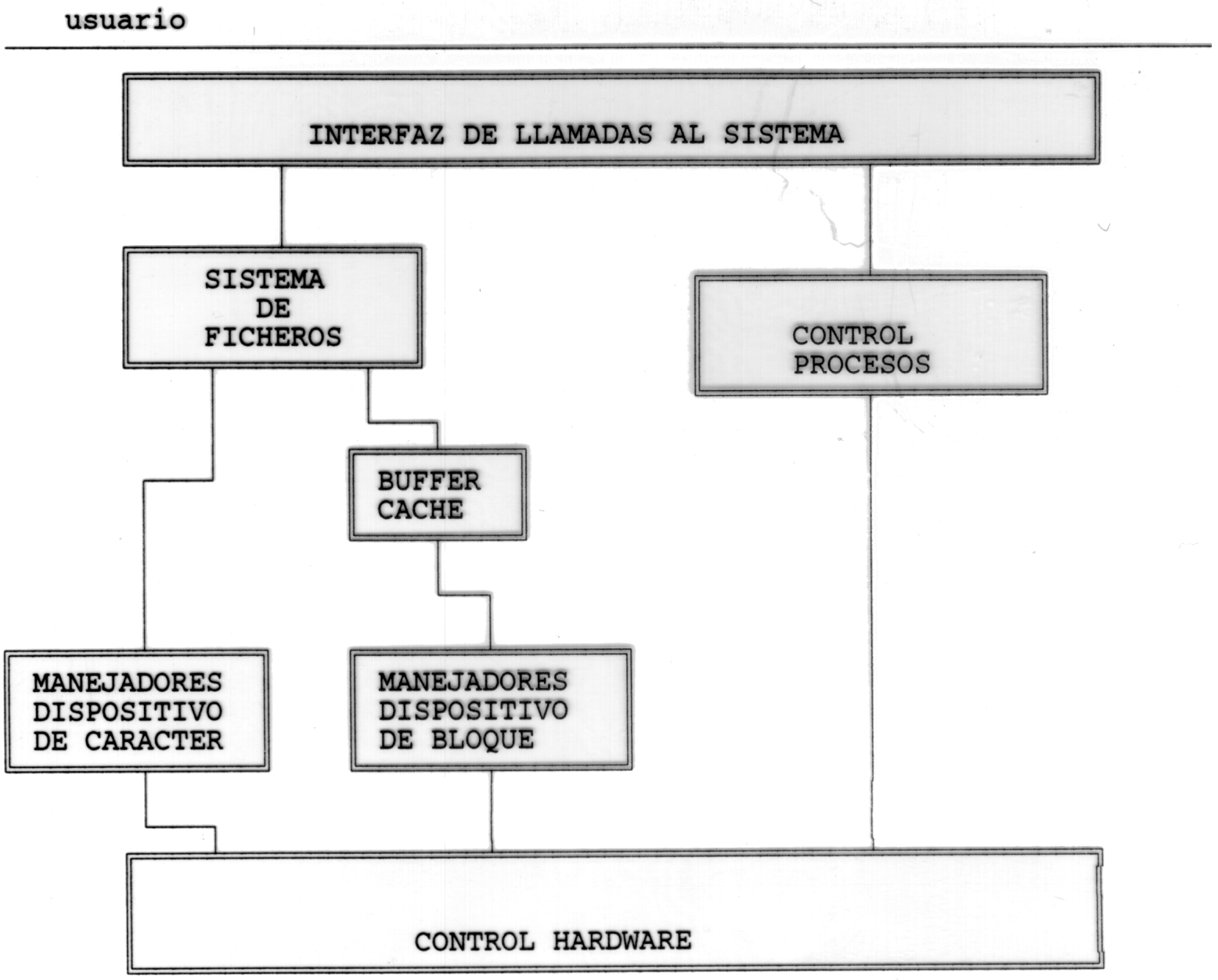
ejemplo de tablas de ficheros



Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

buffer cache



estructura del buffer cache

- Estructura software para minimizar accesos a disco
- Sistema de buffers de datos con los bloques de disco mas recientemente
  - el kernel necesita leer datos: leer del buffer cache
  - el kernel necesita escribir datos: escribe en el buffer cache
  - un bloque de disco solo puede estar en un solo buffer
- formado por una serie de buffers organizados en dos estructuras
  - lista de buffers libres (FREELIST). En cada instante contiene los buffers que no están siendo utilizados en ese momento. Se utiliza para matener el orden de reemplazo LRU. Un buffer del cache estará o no estará en la FREELIST en un instante dado dependiendo de si está siende utilizado o no en ese instante. Usada para mantener orden LRU en reemplazo
  - array de colas hash. Los buffers se organizan en el cache en una array de colas hash para acelerar las búsquedas. La función hash es una función hash de la identificación del buffer (número de dispositivo y número de bloque)

estructura de un buffer

- cada buffer contiene, aparte de los datos de un bloque de disco, una cabecera con información diversa
- entre la información mas relevante en la cabecera
  - identificación del buffer (la del bloque que contiene
  - punteros para mantener las diversas estructuras (cola hash, free list ...)
  - estado del buffer, que incluye
    - ocupado
    - datos válidos
    - modificado (*delayed write*)
    - pendiente de e/s
    - pocesos en espera por este buffer

buffer en openBSD 4.0

```
/*
 * The buffer header describes an I/O operation in the kernel.
 */
struct buf {
   LIST_ENTRY(buf) b_hash;          /* Hash chain. */
   LIST_ENTRY(buf) b_vnbufs;        /* Buffer's associated vnode. */
   TAILQ_ENTRY(buf) b_freelist;     /* Free list position if not active. */
   TAILQ_ENTRY(buf) b_synclist;     /* List of dirty buffers to be written out */
   long b_synctime;                 /* Time this buffer should be flushed */
   struct  buf *b_actf, **b_actb;   /* Device driver queue when active. */
   struct  proc *b_proc;            /* Associated proc; NULL if kernel. */
   volatile long   b_flags;         /* B_* flags. */
   int     b_error;                 /* Errno value. */
   long    b_bufsize;               /* Allocated buffer size. */
   long    b_bcount;                /* Valid bytes in buffer. */
   size_t  b_resid;                 /* Remaining I/O. */
   dev_t   b_dev;                   /* Device associated with buffer. */
   struct {
           caddr_t b_addr;          /* Memory, superblocks, indirect etc. */
   } b_un;
   void    *b_saveaddr;             /* Original b_addr for physio. */
   daddr_t b_lblkno;                /* Logical block number. */
   daddr_t b_blkno;                 /* Underlying physical block number. */
                                    /* Function to call upon completion.
                                     * Will be called at splbio(). */
   void    (*b_iodone)(struct buf *);
   struct  vnode *b_vp;             /* Device vnode. */
   int     b_dirtyoff;              /* Offset in buffer of dirty region. */
   int     b_dirtyend;              /* Offset of end of dirty region. */
   int     b_validoff;              /* Offset in buffer of valid region. */
   int     b_validend;              /* Offset of end of valid region. */
   struct  workhead b_dep;          /* List of filesystem dependencies. */
};
```

flags buffer en openBSD 4.0

```
/*
 * These flags are kept in b_flags.
 */
#define B_AGE           0x00000001      /* Move to age queue when I/O done. */
#define B_NEEDCOMMIT    0x00000002      /* Needs committing to stable storage */
#define B_ASYNC         0x00000004      /* Start I/O, do not wait. */
#define B_BAD           0x00000008      /* Bad block revectoring in progress. */
#define B_BUSY          0x00000010      /* I/O in progress. */
#define B_CACHE         0x00000020      /* Bread found us in the cache. */
#define B_CALL          0x00000040      /* Call b_iodone from biodone. */
#define B_DELWRI        0x00000080      /* Delay I/O until buffer reused. */
#define B_DIRTY         0x00000100      /* Dirty page to be pushed out async. */
#define B_DONE          0x00000200      /* I/O completed. */
#define B_EINTR         0x00000400      /* I/O was interrupted */
#define B_ERROR         0x00000800      /* I/O error occurred. */
#define B_GATHERED      0x00001000      /* LFS: already in a segment. */
#define B_INVAL         0x00002000      /* Does not contain valid info. */
#define B_LOCKED        0x00004000      /* Locked in core (not reusable). */
#define B_NOCACHE       0x00008000      /* Do not cache block after use. */
#define B_PAGET         0x00010000      /* Page in/out of page table space. */
#define B_PGIN          0x00020000      /* Pagein op, so swap() can count it. */
#define B_PHYS          0x00040000      /* I/O to user memory. */
#define B_RAW           0x00080000      /* Set by physio for raw transfers. */
#define B_READ          0x00100000      /* Read buffer. */
#define B_TAPE          0x00200000      /* Magnetic tape I/O. */
#define B_UAREA         0x00400000      /* Buffer describes Uarea I/O. */
#define B_WANTED        0x00800000      /* Process wants this buffer. */
#define B_WRITE         0x00000000      /* Write buffer (pseudo flag). */
#define B_WRITEINPROG   0x01000000      /* Write in progress. */
#define B_XXX           0x02000000      /* Debugging flag. */
#define B_DEFERRED      0x04000000      /* Skipped over for cleaning */
#define B_SCANNED       0x08000000      /* Block already pushed during sync */
#define B_PDAEMON       0x10000000      /* I/O started by pagedaemon */
```

buffer cache

- todo buffer está en una cola hash. Cuando sus contenidos son reemplazados, cambia a la nueva cola hash que le corresponde según la identificación del bloque que contiene
- un buffer puede o no estar en la FREELIST.
  - cuando se mete un buffer en la FREELIST, se hace por el final para preservar orden LRU de reemplazo (salvo casos execepcionales)
  - Cuando se saca un buffer de la FREELIST
    - si lo que se quiere es un buffer para ser reemplazado: se saca el primero
    - si se quiere un buffer concreto: se saca dicho buffer independientemente de donde esté

algoritmos del buffer cache

El funcionamiento del buffer cache se describe en estos 4 algoritmos

- **getblk**: Obtiene un buffer para un bloque. No contiene necesariamente los datos del bloque. Usado por *bread*. En algunos casos puede implicar una escritura a disco
- **bread**: Devuelve un buffer con los datos del bloque de disco solicitado. Usado, entre otras llamadas, tanto por la llamada al sistema *read* como por la llamada al sistema *write*. No implica necesariamente una lectura de disco.
- **bwrite**: Escribe un buffer del cache a disco. Usado, por ejemplo, en la llamada al sistema *umount*
- **brelse**: Libera un buffer, marcándolo como libre y colocándolo en la FREELIST

getblk

- entrada: identificación de un bloque de disco
- salida: buffer para ese bloque; no contiene necesariamente los datos del bloque: se indica con la marca de *datos validos*
- varias posibilidades
  a. el buffer buscado está en la cola hash que le corresponde y además el buffer está libre (en la FREELIST)
  b. el buffer buscado está en la cola hash que le corresponde pero está ocupado
  c. el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache)
  d. el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache). Además el primer buffer de la FREELIST está marcado modificado (*delayed write*)
  e. el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache). Además la FREELIST está vacia.

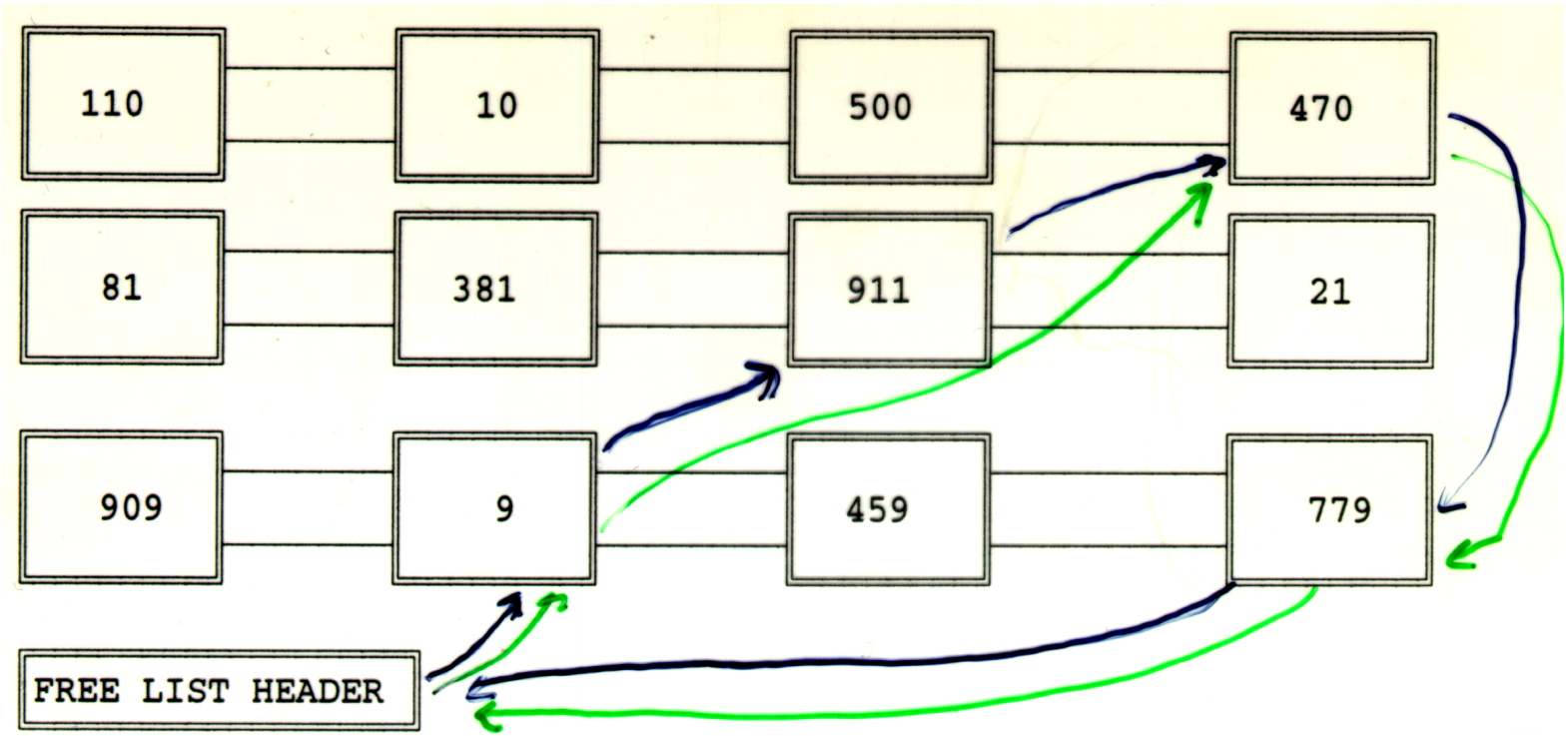
getblk

a. el buffer buscado está en la cola hash que le corresponde y además el buffer está libre (en la FREELIST)
   - marca buffer ocupado
   - quita buffer de la FREELIST
   - devuelve buffer (marca datos válidos)

b. el buffer buscado está en la cola hash que le corresponde pero está ocupado
   - proceso queda en espera hasta que buffer libre, marca buffer como demandado (*wanted*)
   - cuando termina la espera vuelve a buscar en cola hash (reinicia el algoritmo)

caso a) de getblk

- freelist antes de ejecutarse *getblk* solicitando bloque 911
- free list después



getblk

c el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache)

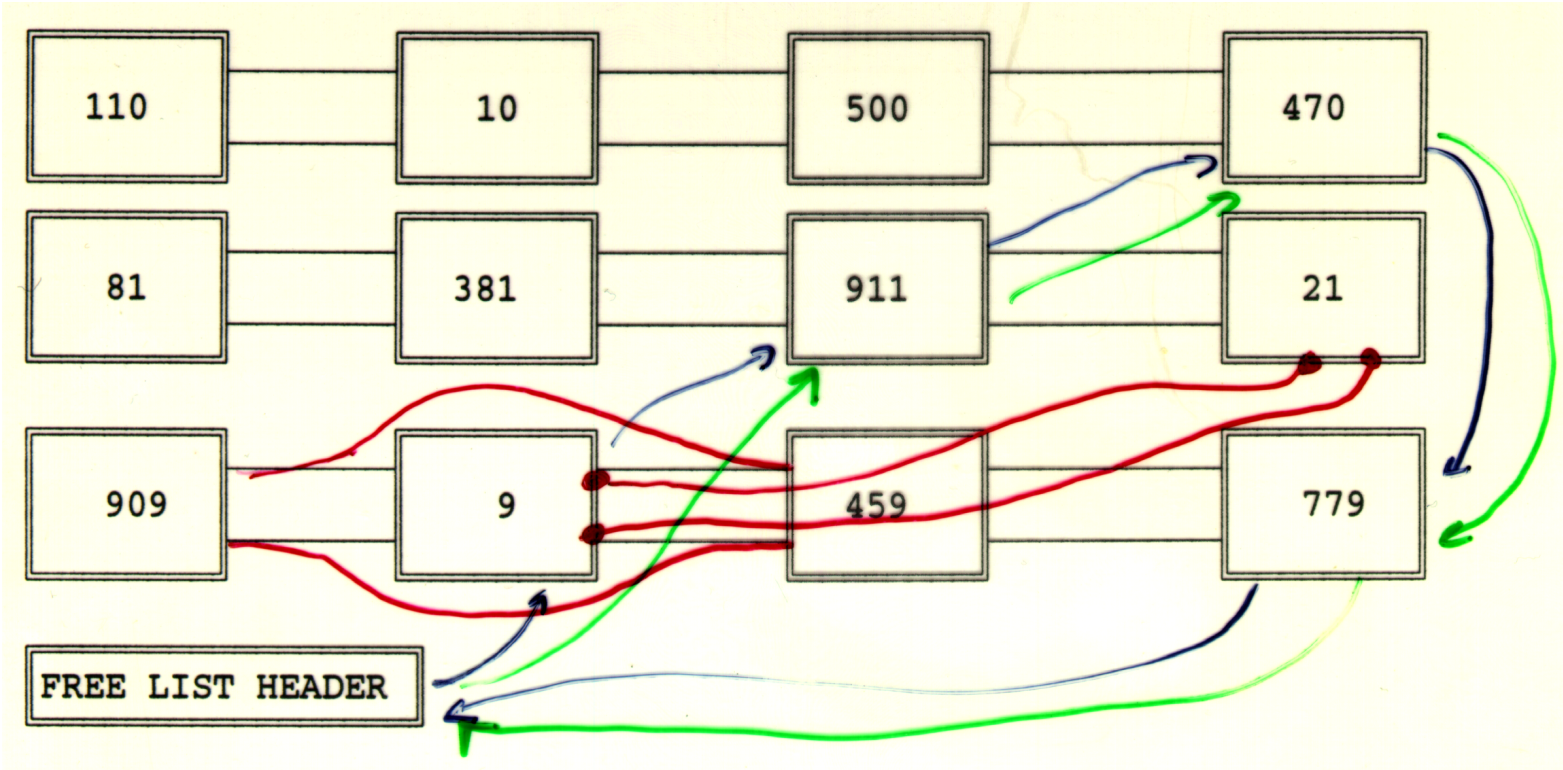
- se toma el primero FREELIST
- se marca como ocupado
- se quita de la FREELIST
- se situa en cola hash correspondiente
- se devuelve (marca datos no válidos)

d el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache). Además el primer buffer de la FREELIST esta modificado (*delayed write*)

- toma el primero de la FREE LIST
- se marca como ocupado
- se quita de la FREELIST: está modificado (marca *delayed write*)
- se inicia la escritura asíncrona en disco
- se vuelve a buscar en cola hash (reinicio del algoritmo)

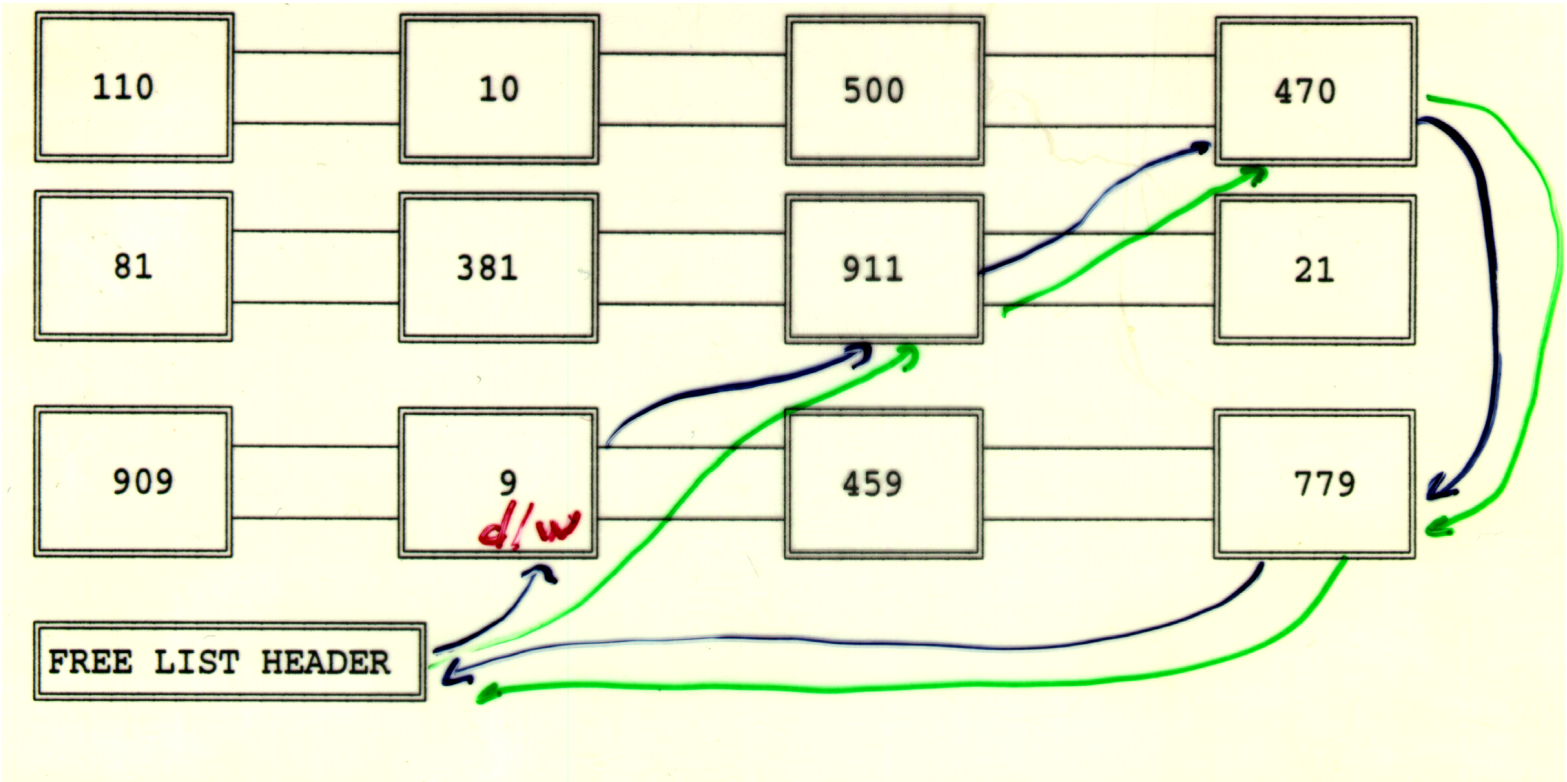
caso c) de getblk

- freelist antes de ejecutarse *getblk* solicitando bloque 451
- free list después
- nueva cola hash



caso d) de getblk

- freelist antes de ejecutarse *getblk* solicitando bloque 451
- free list después
- el bloque 9 se escribe a disco. Se reinicia el algoritmo



getblk

- e el buffer buscado no está en la cola hash que le corresponde (y por tanto no está en el cache). Además la FREELIST está vacia.
  - proceso queda en espera hasta que haya algun buffer libre (FREELIST no vacia)
  - cuando termina la espera vuelve a buscar en cola hash (reinicia el algoritmo)
- procesos compiten por buffers
- al liberar un buffer se despierta a TODOS los procesos que esperaban por ese buffer y tambien a TODOS los que esperaban a que la FREELIST no estuviese vacía
- al volver de una espera deben reiniciar *getblk*

pseudocódigo getblk

```
algoritmo getblk
  entrada: identificacion bloque
  salida: buffer ocupado
{
  while (no se haya hecho)
  {
    if (bloque en cola hash)
    {
      if (bloque ocupado)
      {
        sleep (haya buffer libre);
        continue;
      }
      marcar buffer ocupado;
      quitar FREE LIST;/*elevar ipl*/
      return (buffer);/*datos validos*/
    }
    else
```

pseudocódigo getblk (continuación)

```
    {
      if (FREE LIST vacia)
      {
        sleep(haya buffer libre);
        continue;
      }
      marcar primer buffer ocupado;
      quitar primer buffer FREE LIST; /*elevar ipl*/
      if (buffer modificado)
      {
        iniciar escritura asincrona;
        continue;
      }
      quitar buffer de su cola hash;
      poner en nueva cola hash;
      marcar datos no validos;
      return (buffer); /*datos no validos*/
    } /*else*/
  } /*while*/
} /*getblk*/
```

pseudocódigo bread

```
algoritmo bread
  entrada: identificacion bloque
  salida: buffer ocupado con datos bloque;
{
  obtener buffer -getblk-;
  if (datos validos)
    return (buffer);
  iniciar lectura disco;
  sleep (hasta que se complete lectura);
  marcar datos validos;
  return (buffer); /*datos validos*/
}
```

pseudocódigo bwrite

```
algoritmo bwrite
  entrada: buffer ocupado
{
  iniciar escritura en disco;
  if (escritura sincrona)
  {
    sleep (hasta que escritura completa);
    liberar buffer -brelse-;
  }
  else
    if (buffer no fue reemplazado reemplazado por tener marca d/
      marcar buffer 'viejo';
}
```

pseudocódigo brelse

```
algoritmo brelse
  entrada: buffer ocupado
{
  despertar procesos esperando por buffer libre;
  if (buffer valido y no viejo)
    poner al final FREE LIST;
  else
    poner al principio FREE LIST;
  marcar buffer libre;
}
```

pseudocódigo breada

```
algoritmo breada  /*block read ahead*/
entrada:  identificacion bloque lectura inmediata
    identificacion bloque lectura asincrona;
salida:  buffer ocupado con datos bloque lectura inmediata;
{
  if (primer bloque no en cache)
  {
    obtener buffer para primer bloque-getblk;
    if (datos buffer no validos)
      iniciar lectura en disco;
  }
  if (segundo bloque no en cache)
  {
    obtener buffer para el segundo bloque-getblk;
    if (datos buffer validos)
      liberar buffer-brelse;
    else
      iniciar lectura de disco;
  }
  if (primer bloque estaba en cache)
  {
    leer primer bloque-bread;
    return (buffer);
  }
  sleep (hasta primer buffer contenga datos validos);
  return (buffer);/*del primer bloque*/
}
```

interacción entre *brelse* y *getblk*

- en el caso de realizar una escritura asíncrona con *bwrite*, *bwrite* no libera el buffer
- el buffer se liberará cuando la escritura se complete, suceso que vendrá marcado por una interrupción de dispostivo
- dado que, por tanto, *brelse* puede ser invocado por una interrupción, las estructuras de datos susceptibles de ser accedidas por *brelse* (en concreto la FREELIST) deben ser protegidas
- es necesario elevar el *ipl* en *getblk* al acceder a la FREELIST, lo suficiente para no atender las interrupciones de disco.

buffer cache: ventajas y desventajas

- ventajas
  - acceso más uniforme a disco: todo el acceso es a través del cache
  - no hay restricciones de alineamiento de datos para los procesos de usuario
  - reducción del tráfico a disco
- inconvenientes
  - acceso más lento para transferencias voluminosas
  - modificaciones sobre el cache, el sistema está en estado inconsistente hasta que las modificaciones se actualizan al disco

Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

- Cada fichero (o directorio, o dispositivo) está representado en disco por una estructura pequeña denominada *inodo*
- Cada unidad lógica contiene una lista de inodos

Sistema de ficheros en UNIX

- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones

Sistema de ficheros en UNIX
- Estructura del sistema de ficheros de unix
- El buffer cache
- Representación interna de ficheros
- Diferencias entre el sistema de ficheros unix System V y unix BSD
- Otras consideraciones