Logical disks

Bach 2.2.1

- Physical disk is divided into partitions or *logical* disks
- Logical disk $\equiv$ linear sequence of fixed size, randomly accessible, blocks
  - disk device driver maps underlying physical storage to logical disks
  - large blocks $\Rightarrow$ faster access, more fragmentation
- *File system* $\equiv$ logical disk whose blocks have been arranged suitably so that files may be created and accessed

NOTE:

1. Logical disk may also be used for swap partition
2. Logical disk may be located on multiple physical disks (e.g. volumes, striping, raid)

File systems

- File $\equiv$ inode (header) + data

Boot block	Super block	Inode list	Data blocks

- Boot block: usually contains bootstrap code
- Super block: size, # files, free blocks, etc.
- Inode list
  - size fixed when configuring file system
  - contains *all* inodes
- Data blocks: file data
- Files, directories organized into tree-like structure

Super block

Bach 4.5

- Size of the file system
- Free data blocks: # of free blocks, list of free blocks, pointer to the first free block on free list
- Size of inode list
- Free inodes: (as above)
- Locks for free block list, free inode list
- Dirty flag

Inodes

Bach 4.1

File attributes:

- Type: regular file, directory, device special file, pipes
- Owner: individual + group
- Permissions: read, write, execute, for owner, group, others
- Access times: last modified, last accessed, last modification of inode
- Number of links
- File size: 1 + highest byte offset written into
- Table of contents: disk addresses of (discontiguous) disk blocks containing the file data

```
-rw-rw-r-   1 mandar mandar    2647   Mar 11 14:58   filesys.tex
```

Table of contents

Bach 4.2

- File data stored in non-contiguous disk blocks
  - contiguous storage $\Rightarrow$ fragmentation, expansion of files problematic
- File space is allocated one block at a time (i.e. data can be spread throughout file system)
- 10 direct entries: numbers of disk blocks that contains file data
- Single indirect: number of a disk block that contains a list of direct block numbers
- Double indirect, triple indirect
- Processes access data by byte offset; kernel converts byte offset into block no.
- Block no. = 0 $\Rightarrow$ corresponding logical block contains no data
  - no disk space is wasted

Directories (SVR2)

Bach 4.3

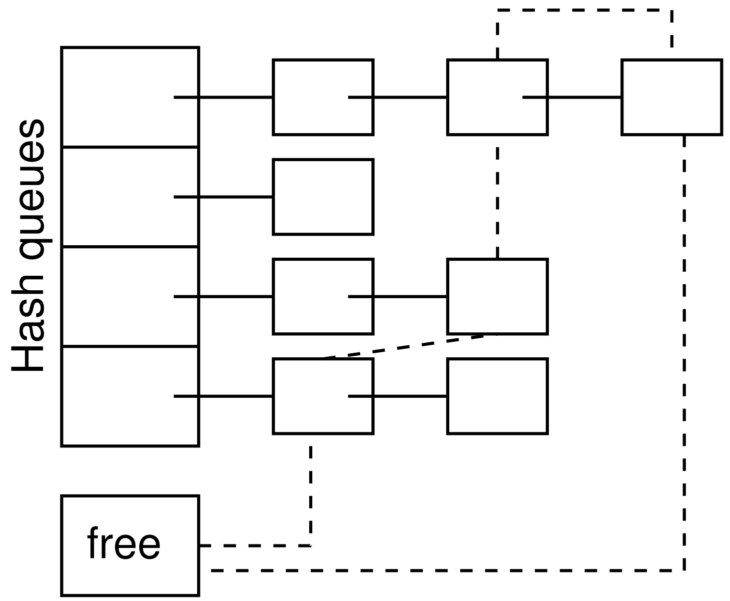
- Data blocks contain a sequence of 16 byte entries
- Each entry = inode number (2 bytes) + null-terminated file names (14 bytes)
- Compulsory entries: current directory (`.`) and parent directory (`..`)
  - for root, parent directory = root
- Inode number = 0 $\Rightarrow$ empty directory entry

Inode cache/table

Bach 4.1

- List of buffers stored in main memory
- Each buffer contains *in-core* copy of *disk* inode
- **At most one copy of any inode is present in the cache**
- Additional information stored in each buffer:
  - logical device number of file system that contains the file
  - inode number
    - inode list on disk $\equiv$ linear array
    - numbering starts from 1
  - pointers to other in-core inodes*
  - status (locked/free/awaited, dirty bit, mount point flag)
  - reference count: # of active uses of the file

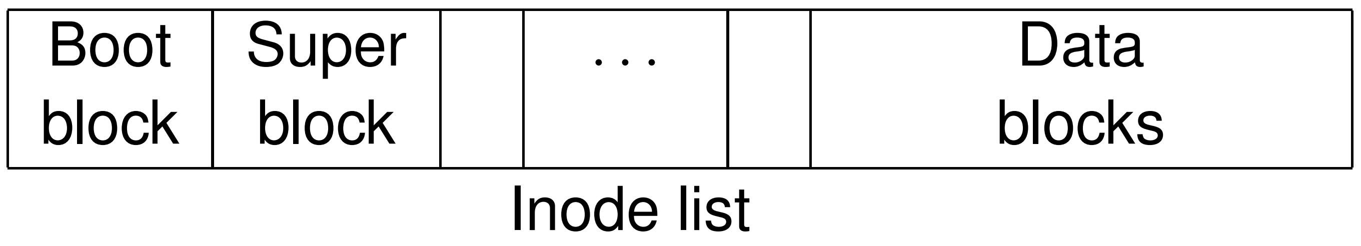
Inode cache



Inode cache

- *Free list:* doubly linked circular list of inodes that are not in active use
  - reference count = 0
$\Leftrightarrow$ inode is on free list
$\Leftrightarrow$ kernel can reallocate buffer to hold another disk inode
  - initially, all buffers are free
  - kernel takes inode buffer from head of list when it needs a free buffer
  - kernel returns buffer to end of list when done
  - buffer allocation policy = LRU
- *Hash queue*
  - buffer pool organized into separate circular, doubly linked lists
  - inode is assigned to a queue using a hash function of $\langle$ dev. no., block no. $\rangle$
- Inode buffer can simultaneously be on free list, hash queue

Accessing inodes



Input: Inode no. i

Output: Location (Block no., byte offset)

Method:

1. Let $n =$ number of inodes per block
2. Block no. $B = (i - 1)$ div n + starting block of inode list
3. Byte offset $b = ((i - 1)$ mod $n) \times$ size of disk inode
4. Return $\langle B, b\rangle$

Algorithms

- *iget*, *iput*
  - get / release a known inode
  - used for opening / closing a file
- *ialloc*, *ifree*
  - allocate / free a new inode
  - used for creating / deleting files
- *alloc*, *free*
  - allocate / free a new disk block
  - used for adding / removing blocks from a file

iget

Input: device no., inode no. **Output:** locked inode

Bach 4.1

Algorithm:

```
while (not done) {
  if (inode in cache) {
     if (inode locked) {
        sleep till inode
          becomes unlocked;
        continue; /* !! */
     }
     remove from free list
       if necessary;
     increment ref_count;
     return inode;
  }
  /* inode not in cache */
  if (free list is empty)
     return error; /* why? */
  remove buffer from free list;
  reassign to correct hash queue;
  read inode from disk;
  initialize ref_count = 1;
  return inode;
}
```

iput

Input: device no., inode no. **Output:** none

Algorithm:

```
lock inode (if not already locked);
decrement ref_count;
if (ref_count = 0) {
   if (link count = 0) {
      free disk blocks for the file;
      set file type = 0; /* type = 0 <=> inode is free */
      free inode; /* ifree() */
   }
   if (file accessed / changed, or inode changed)
      update disk copy; /* since in-core copy is "dirty" */
   put inode on free list;
}
release lock;
```

Locks vs. reference count

- Locks are used to ensure mutual exclusion
- Locks **<u>never</u>** last across system calls
    - always released at the end of a system call
- Reference count = number of active uses
- Reference count remains set across multiple system calls
    - prevents kernel from reassigning an inode that is in use

ialloc

Bach 4.6

- Superblock has a cache containing the numbers of free inodes (type = 0)

Algorithm:

```
while (!done) {
  if (SB is locked) {
     sleep until SB is free;
     continue;
  }
  if (inode list in SB is empty) {
     lock SB;
     starting from "remembered inode", search disk for free inodes;
     add free inodes to SB list until full, or no more free inodes;
     update "remembered inode" for next search;
     unlock SB; /* wakes up other procs */
     if (no free inodes) return error;
  }
```

ialloc

```
  get inode from SB inode list;
  get corresponding inode from inode cache; /* iget */
  if (inode is not free) { /* race condition! */
     release inode; /* iput() */
     continue;
  }
  initialize inode;
  write inode to disk;
  decrement free inode count;
  return inode;
}
```

ialloc

Race condition:

1. Kernel assigns inode I to process P_A; P_A goes to sleep before reading disk copy into memory.
2. P_B needs inode, finds list is empty, searches for free inodes, finds I is free, puts I on free list.
3. P_A wakes up, initializes I and uses it.
4. When P_C requires inode, gets I, but I is not free!

ifree

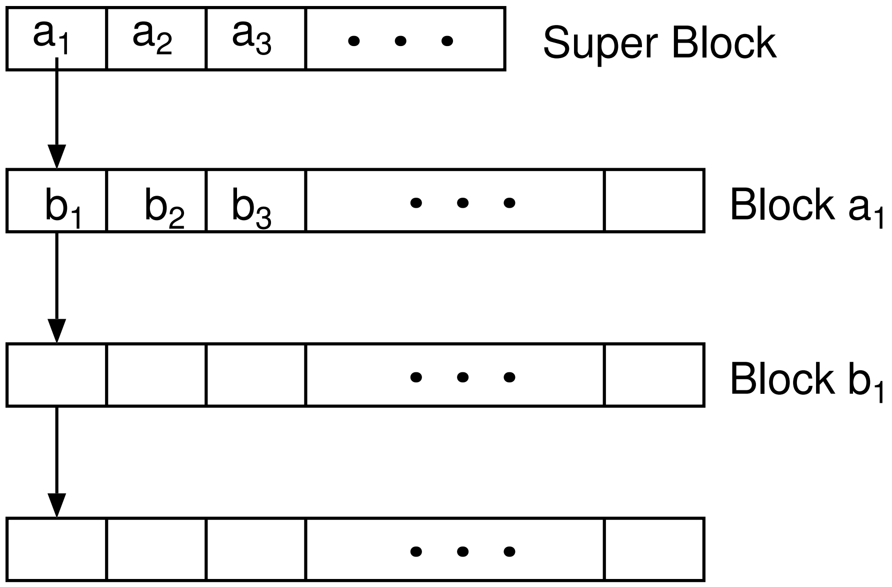
Algorithm:

```
increment file system free inode count;
if (SB is locked) return;
lock SB;
if (inode list is full) {
   if (inode no. < "remembered inode")
      update "remembered inode";
}
else store inode no. in free inode list;
unlock SB;
```

Disk block allocation

- SB contains a list of free disk block nos.
- Initially, `mkfs` organizes all data blocks in a linked list
  - each link (disk block) contains (i) list of free disk block nos., (ii) no. of next block on the list

Bach 4.7



- Free nodes identifiable by type field; free disk blocks not identifiable by content
- Disk blocks consumed more quickly than inodes
- Disk blocks large enough to contain long list of free block nos.

alloc

Algorithm:

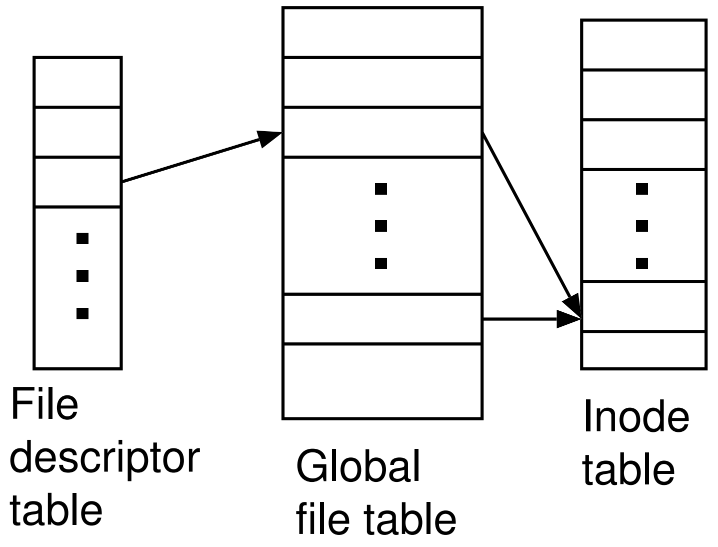
```
while (SB is locked) sleep until SB is free;
remove block from SB free list;
if (last block was removed) {
   lock SB;
   read block just removed;
   copy block nos. into SB list;
   unlock SB; /* wake up other procs */
}
zero block contents;
decrement total count of free blocks;
mark SB modified;
return buffer;
```

free

Algorithm:

```
if (SB list is not full)
   put block on SB list;
if (SB list if full) {
   copy SB list into freed block;
   write block to disk;
   put block no. of freed block into SB list;
}
```

System calls



- File descriptor (per process) – pointers to all open files
- Global file table – mode, offset for each `open`-ed file
- Inode table – memory copy of on-disk inode (only one per file)

- *creat*, *open*, *close*
- *read*, *write*
- *mount*, *umount*

open

Syntax:

```
fd = open(pathname, flags, mode);
flags = O_RDONLY, O_RDWR, etc.
mode - used only if file is created
```

Algorithm:

1. Find in-core inode for given filename (pathname lookup, followed by *iget*).
2. If file does not exist, or access denied, return error.
3. Allocate and initialize global file table entry.
   - pointer to inode, mode, ref. count, offset (0 or file size)
4. Allocate user fd table entry; set pointer to global file table entry.
5. If file needs to be truncated, free all file blocks (*free*).
6. Unlock inode, return descriptor.

Bach 5.1

close

Syntax: `close(fd);`

Algorithm:

1. Set user fd table entry to `NULL`.
2. Decrement ref. count of global file table entry.
3. If ref. count is 0: free file table entry, release inode (*iput*).

Bach 5.6

creat

Syntax: `fd = creat(pathname, modes);`

Action:

- If file non-existent: new file with specified permissions is created
  (parent directory must have write permission)
- If file existed, file is truncated and opened in write mode
  (file itself should have write permission, parent directory permissions not checked)

Bach 5.7

Algorithm:

1. Parse given pathname.
   - remember inode of parent directory in *u area*, keep inode locked
   - note byte offset of first empty directory slot in the directory and save this in the *u area*
2. If file already exists, and permissions are improper: release inode (*iput*), return error.

creat

4. Otherwise:
   - 4.1 assign free inode from file system (*ialloc*).
   - 4.2 initialize new directory entry in parent directory with new name and inode no.
   - 4.3 write directory with new name to disk.
   - 4.4 release inode of parent directory (*iput*).
5. Allocate file table entry for inode, initialize ref. count.
6. If file existed, free all disk blocks (*free*). (Owner and permissions of old file are retained.)
7. Unlock inode, return file descriptor.

NB: Order of writes is important:

- system crash $\Rightarrow$ allocated inode will not be reachable
- if writes were done in reverse order, system crash $\Rightarrow$ path name would refer to bad inode

read

Syntax: `num_read = read(fd, buffer, num_bytes);`

Algorithm:

1. Get file table entry from user fd.
2. Check file mode (read / write).
3. Set parameters in the *u area*: (i) mode = read (ii) number of bytes to read (iii) offset in file (iv) target memory address (v) flag to indicate that target is in user memory
4. Get inode from file table, lock inode.
5. While target is not reached:
   5.1 convert file offset into $\langle$ disk block no., offset in block $\rangle$.
   5.2 calculate no. of bytes to read; return EOF if necessary.
   5.3 read block into system buffer; copy data from system buffer to user address.
   5.4 update *u area* fields.

read

6. Unlock inode.
7. Update file table offset for next read.
8. Return total no. of bytes read.

Notes:

- If a process reads data from a "non-existent" block, kernel returns null (zero) bytes
- Inode is locked for the duration of the call to ensure that a single `read` call returns consistent data (otherwise, single `read` can return mix of old and new data)
- Inode unlocked at the end of the call $\Rightarrow$ concurrent reading and writing may return mix of data
- `fd1 = open("abc.txt", ...);`
  `fd2 = open("abc.txt", ...);`
  `fd1`, `fd2` are manipulated indenpendently

write

Syntax: `num_written = write(fd, buffer, count);`

Algorithm: as for *read*

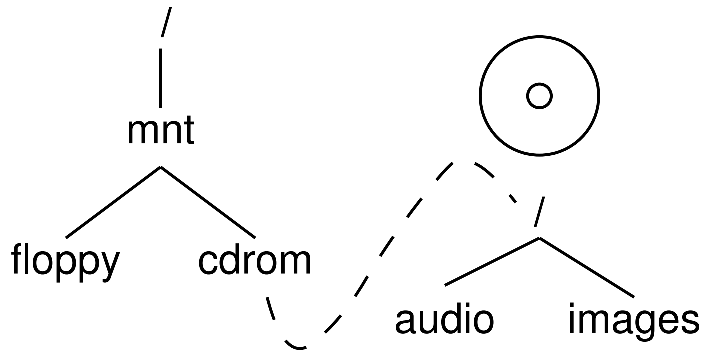
Notes:

- When writing a block:
  - if only a part of the block is written, block must first be read from disk
  - if entire block is written, block need not be read
- If the file does not contain a block corresponding to the byte offset to be written, kernel allocates a new block (*alloc*); fills in ToC slot with this block no.
  - multiple blocks may have to be allocated if the offset corresponds to an indirect block

Bach 5.3

mount

Bach 5.14



Syntax:

```
mount(dev, dir, options);
dev = dev.  spl.  file for fs to be mounted
dir = mount point
options = read-only, etc.
```

Mount table: 1 entry for each mounted file system

- device no.
- pointer to buffer containing super block of mounted f.s.
- pointer to root inode of mounted f.s.
- pointer to inode of mount point directory

mount

Algorithm:

1. If not super user, return error.
2. Get inode for block special file corresponding to the file system to be mounted; extract major and minor numbers.
3. Get inode for mount point directory. If (not directory or ref. count > 1), release inodes; return error.
4. Find free slot in mount table; mark slot in use; initialize device # field.
   - process could go to sleep in ensuing read; marking slot prevents other process from using the same M.T. entry.
   - recording device # prevents other processes from mounting same file system again.
5. Call block device *open* routine (legality checks, initialization of hardware and driver data structures, etc.).

mount

6. Read SB into system buffer; initialize SB fields.
   - locks are cleared
   - no. of free inodes in SB list is set to 0
     - minimizes chance of file system corruption when mounting f.s. after a crash, since *ialloc* scans disk and constructs an accurate list of free inodes
7. Get root inode of mounted f.s. (*iget*); save pointer in mount table.
8. Set `mount_point` flag in inode of mount point directory; save pointer to this inode.
   - allows path names to use `..` to traverse mount point directory
9. Release special file inode (*iput*); unlock inode of mount point directory.

umount

Syntax: `umount(dev);`

Algorithm:

1. If not superuser, return error.
2. Get inode of device special file; extract major and minor nos. of device being unmounted; release inode of special file.
3. Get mount table entry, based on major/minor #.
4. Check whether files on the f.s. are still in use:
   - 4.1 search inode table for all files whose dev. # matches the f.s. to be unmounted;
   - 4.2 if any such file has ref. count > 0 (current directory of some process / open files that have not been closed), return error.
5. Update SB, inodes, flush any unwritten data.
6. Get root inode of mounted f.s. from M.T.; release inode (*iput*).

umount

7. Invoke *close* routine for device.
8. Get and lock inode of mount point via M.T.
9. Clear `mount_point` flag, release inode *iput*.
10. Free system buffer used for SB; free M.T. entry.

dup

Syntax: `newfd = dup(fd);`

Algorithm:

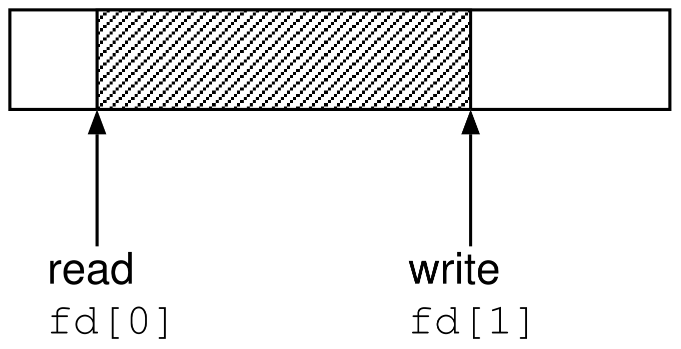
1. Find *first* free slot in the user fd table.
2. Copy given file descriptor into the free slot.
3. Increment ref. count of corresponding global file table entry.
4. Return descriptor (index) of this slot.

Bach 5.13

Pipes

Definition: “pseudo file” with a maximum size which has two file descriptors

Syntax:
```
int fd[2];
status = pipe(fd);
```



Bach 5.12

Shell

Bach 7.8

Algorithm:

1. Parse command line.
2. If command is internal command, call suitable function.
3. If command is external command, fork a child. In child:

    3.1 if input / output redirection is required:

    ```
    fd = /* create new file */
    close(stdout);   dup(fd);   close(fd);
    ```

    3.2 if pipes are required:

    3.2.1 create a pipe; fork a child

    3.2.2 in child: setup pipes s.t. stdout goes to pipe, exec the first component of command line
    in parent: setup pipes s.t. stdin comes from pipe.

    3.3 exec command (or last component of command).
4. If command is run in foreground, wait for child to exit.