The Second Extended File System
Current State, Future Development

Rémy Card
Remy.Card@masi.ibp.fr
Laboratoire MASI, Institut Blaise Pascal
Université Pierre et Marie Curie (Paris VI)

Outline

- History of Linux File Systems
- The Virtual File System
- Features of the Ext2 File System
- Future Ext2 FS Development in Linux 1.3
- Conclusion

The Minix filesystem

- First Linux filesystem
- Implemented by Linus Torvalds when Linux was being cross-developed under Minix
- Simple filesystem:
  - strong limitations:
    - 14 characters file names
    - Maximal size: 64 MB
  - very similar to the Unix V7 FS (but uses bitmaps instead of linked lists)
- Linux extension: supports 30 characters file names (incompatible with Minix)

The Extended File System

- Second Linux filesystem (based on the Minix FS kernel code), released in April 1992
- Implemented by Rémy Card
- Less limitations than the Minix FS
- But: poorly designed free space management:
  - Free blocks are stored in a list
  - The FS tends to become fragmented after some use
  - Slower than the Minix FS
- FS defragmenter written by Stephen Tweedie
- Ext fs is now obsolete

The Xia File System

- Designed and implemented by Frank Xia as a simple and efficient filesystem (based on the Minix FS kernel code), released in January 1993
- New algorithms:
  - Variable length directory entries
  - Bigger control structures (disk addresses, inodes)
  - optimized allocation routines
- Features very similar to the Extended File System’s ones
- But better performances than Ext fs (uses bitmaps instead of linked lists to keep track of free blocks)

The Second Extended File System

- Designed by Rémy Card and Wayne Davison after a "Request for Ideas" on the KERNEL channel
- Implemented by Rémy Card (based on the Ext fs kernel code with many changes...), released in January 1993
- Enhanced by several other programmers (especially Stephen Tweedie for the kernel code and Theodore Ts'o for the utilities)
- Used by most Linux distributions

Summary

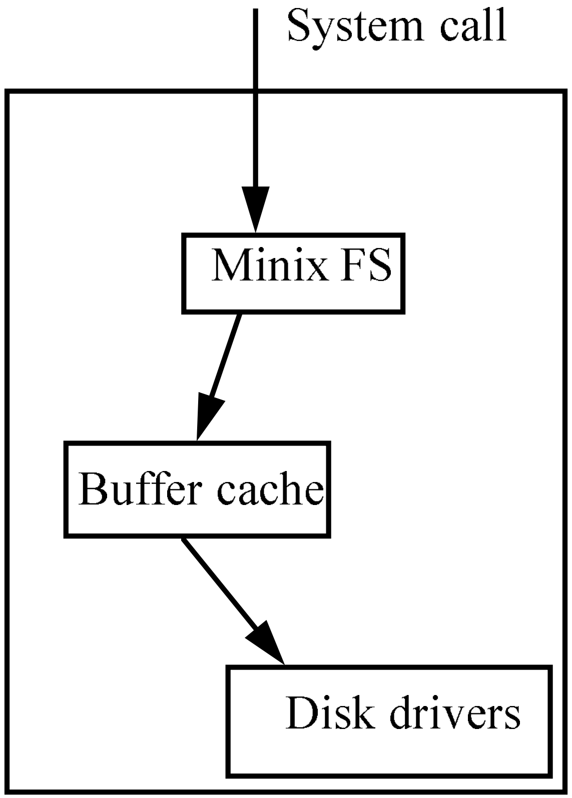
	Minix FS	Ext FS	Ext2 FS	Xia FS
Max FS size	64 MB	2 GB	4 TB	2 GB
Max file size	64 MB	2 GB	2 GB	64 MB
Max file name	14/30 c	255 c	255 c	248 c
3 times support	No	No	Yes	Yes
Extensible	No	No	Yes	No
Var. block size	No	No	Yes	No
Maintained	Yes	No	Yes	??

Principle of the Virtual File System (1)

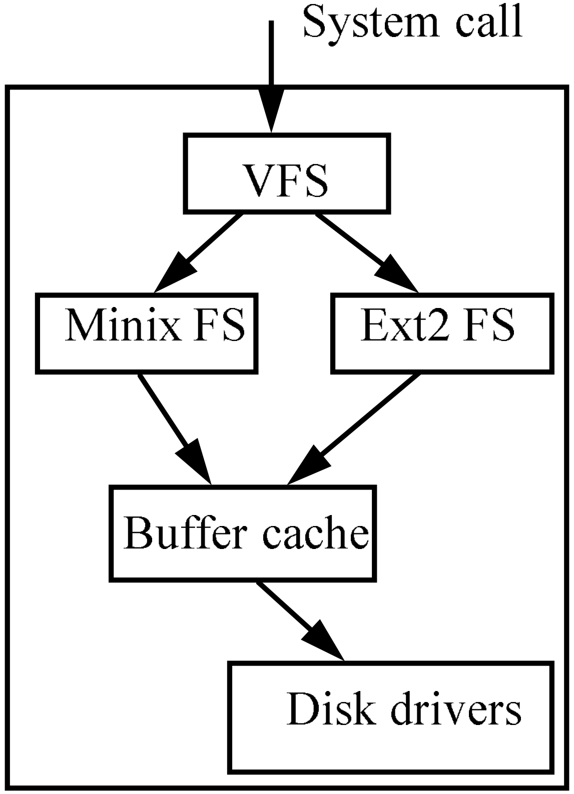
- The Linux kernel contains a Virtual File System layer
- File oriented system calls are directed to the VFS
- These calls are redirected to the appropriate FS modules
- Generic structures:
  - superblocks
  - inodes
  - files
- Function pointers:
  - superblock operations
  - inode operations
  - open file operations

Principle of the Virtual File System (2)

- Before the VFS



- With the VFS



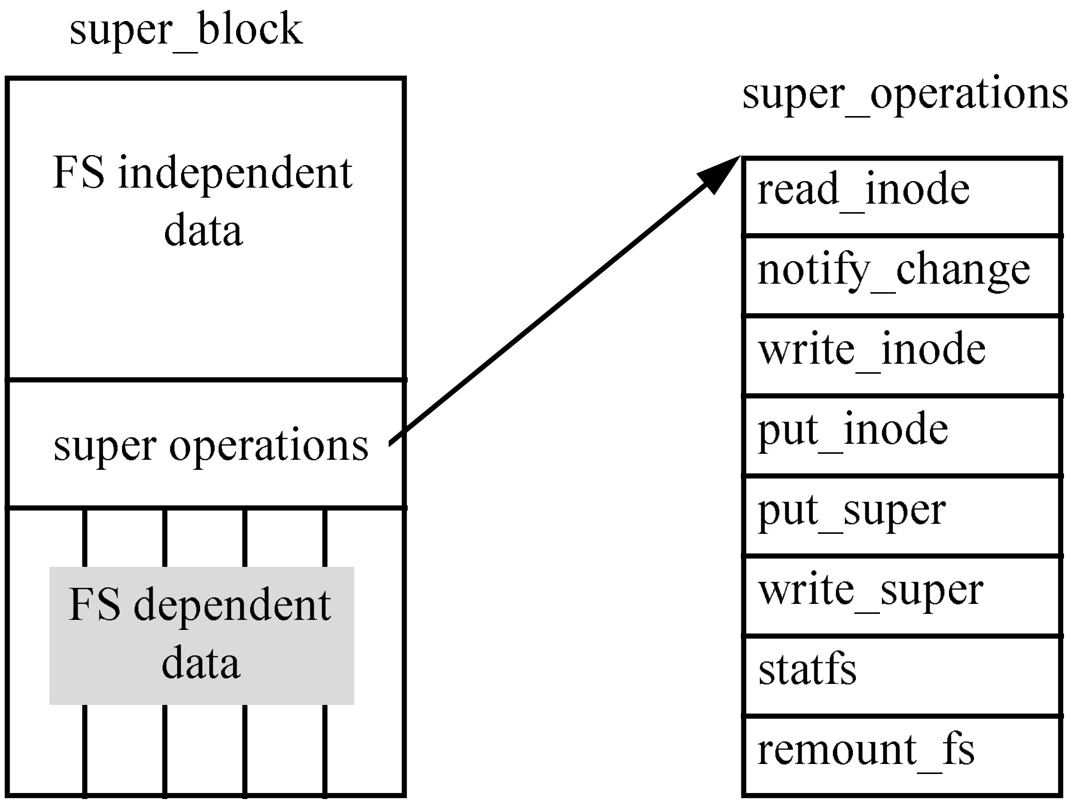
Filesystem types

- Each FS type is described by a structure:
  - name of the FS type
  - mount function
- When a mount request is made, the VFS calls the FS mount function (**`read_super (super, data, silent)`**)
- The FS mount function is responsible for:
  - initializing the FS internal structures
  - reading the FS metadata
  - initializing the FS operation table

Mounted filesystems (1)

- The VFS maintains a table of mounted FS
- Mounted FS are referenced by the structure **`super_block`**:
  - FS independent data
  - pointer to inode covered by this FS
  - pointer to the root inode of the FS
  - pointer to FS dependent operations
  - FS dependent data
- The contents of this structure is initialized at mount time by the FS dependent mount function

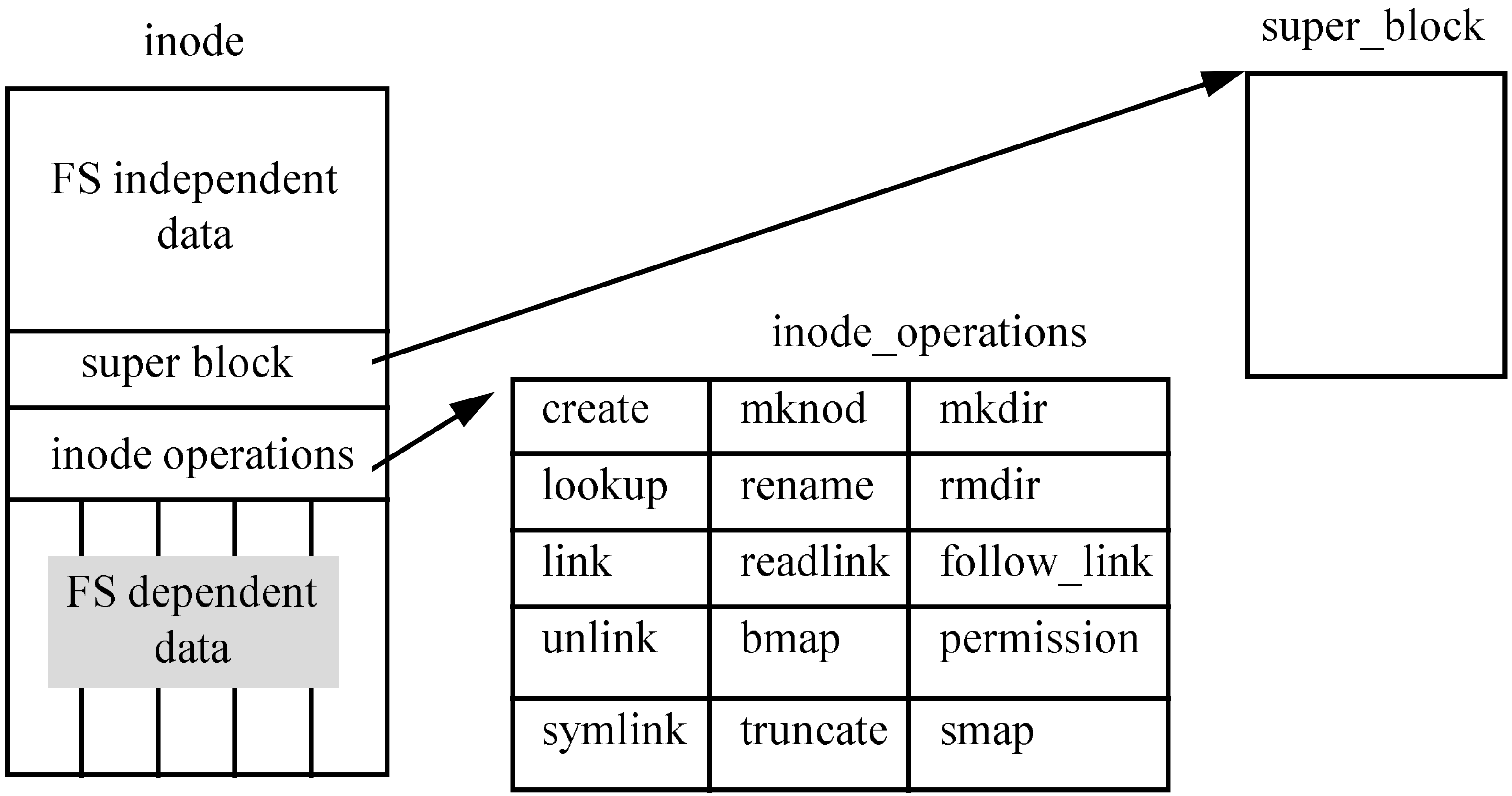
Mounted filesystems (2)



Inodes in use (1)

- Every file operation is made on an inode
- The kernel translates file pathnames into inode numbers
- The VFS maintains a table of inodes in use
- Inodes are referenced by the structure **`inode`**:
  - FS independent data
  - pointer to FS dependent operations
  - FS dependent data
- Usually one operation table per inode type (regular file, directory, symbolic link, ...)

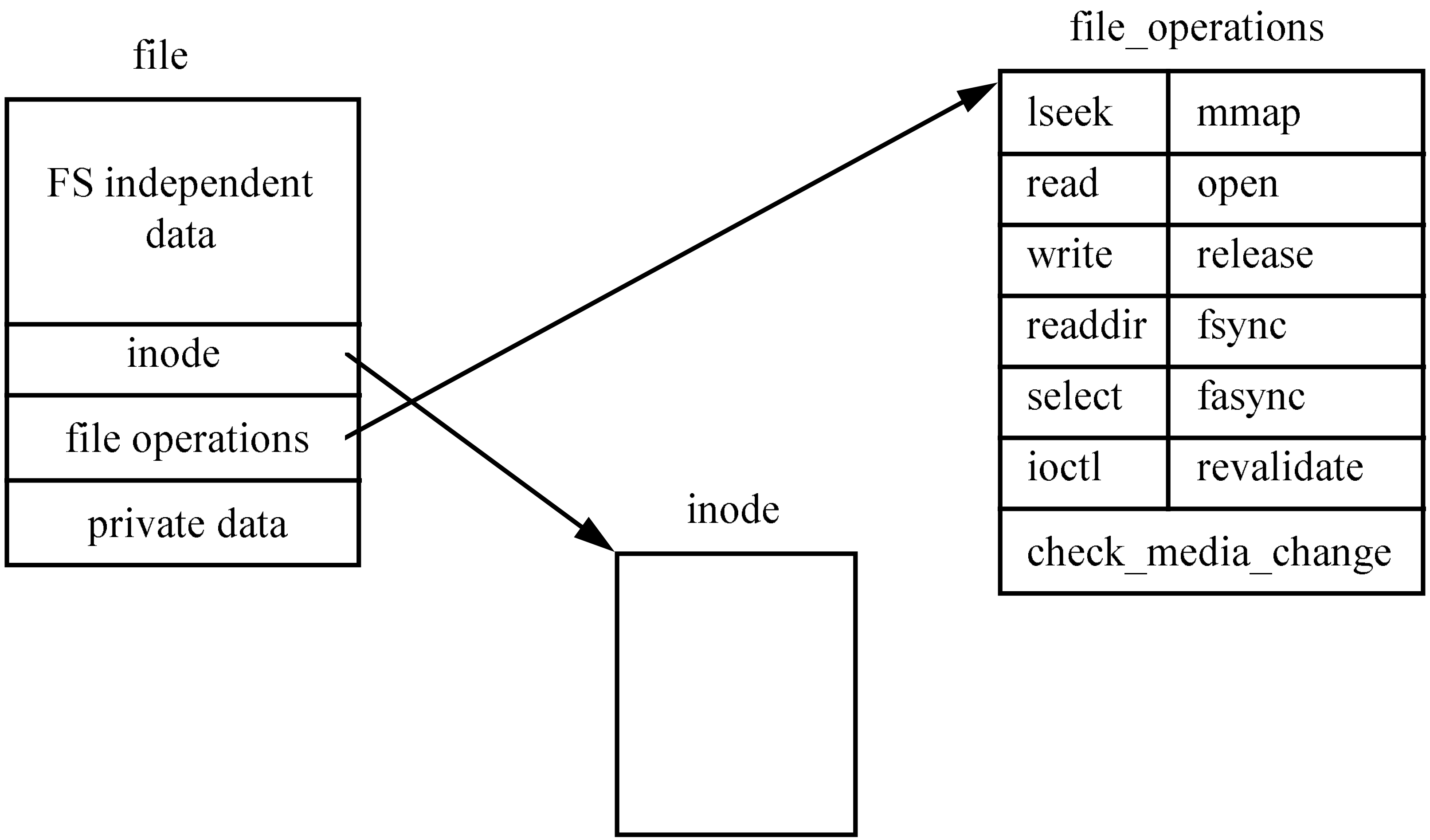
Inodes in use (2)



Open files (1)

- Some system calls act on open files
- Each open file is associated to an inode
- Open files are referenced by the structure **`file`**:
  - FS independent data
  - pointer to the associated inode
  - pointer to FS dependent operations
  - private data

Open files (2)


file
FS independent data
inode
file operations
private data
inode
file_operations
lseek
mmap
read
open
write
release
readdir
fsync
select
fasync
ioctl
revalidate
check_media_change

Features of the Ext2 File System

- Standard Unix features
- Mount options
- Attributes
- Tunable parameters
- Fast symbolic links

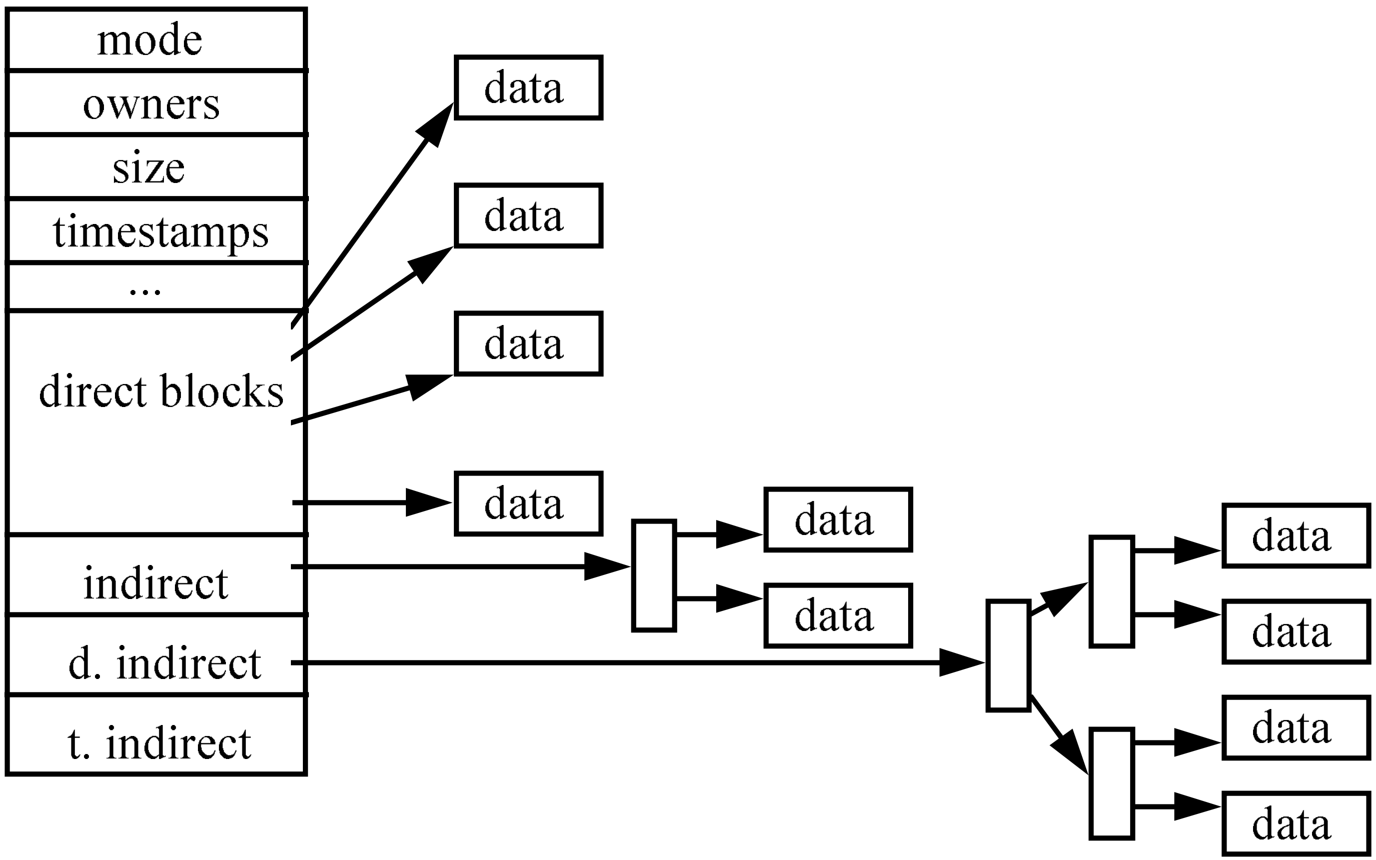
“Standard” Ext2fs features

- Unix files:
  - Regular files
  - Directories
  - Device special files
  - Symbolic links
- Big partitions: up to 4 TB
- File size: up to 2 GB
- Long file names (up to 255 characters)
- Reserved blocks for **`root`** (defaults to 5%)

Inodes in Ext2fs (1)

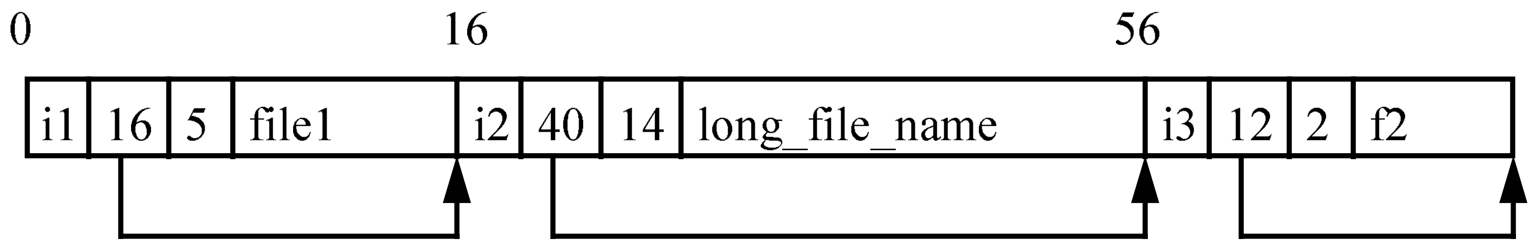
- Each file is represented by an “inode”
- An inode contains the description of the file:
  - mode (file type, access rights)
  - owners (uid, gid)
  - times (access, data modification, inode modification)
  - size
  - pointers to data blocks
  - ...
- Data blocks addresses are kept in the inode
- Inodes are grouped into an inode table

Inodes in Ext2fs (2)



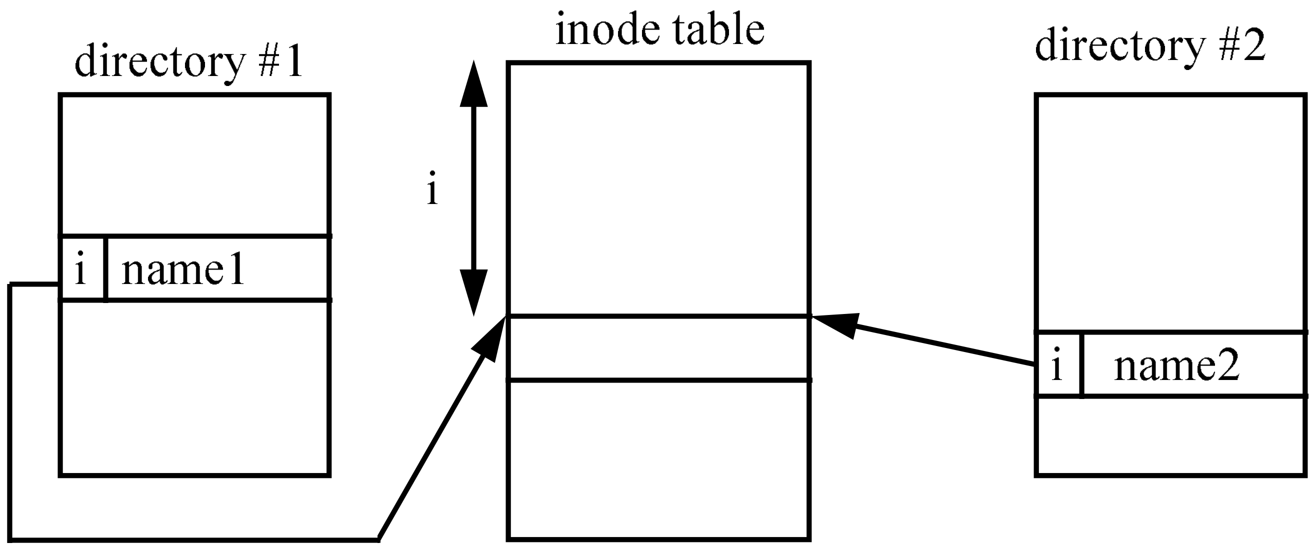
Directories in Ext2fs

- Directory: list of variable length entries
- Each entry contains:
  - the inode number
  - the entry length
  - the file name length
  - the file name
- Example



Links in Ext2fs (1)

- Two types of links: hard links and symbolic links
- Hard link:
  - additional directory entry associated with an inode
  - each inode contains a links count
  - some restrictions: no cross-filesystems hard links, no directory hard link



Links in Ext2fs (2)

- Symbolic link:
  - file which contains a filename
  - when a symbolic link is encountered during a pathname to inode translation, the kernel restarts the interpretation with the link contents
  - no restrictions
  - but: uses more space (inode + data blocks) than a hard link and slows down pathname resolutions

Device files in Ext2fs

- Character or block special files
- No disk space associated with special files
- I/O requests are forwarded to a (pseudo) device driver:
- Referenced in the kernel by:
  - a major number (identifies the driver)
  - a minor number (identifies the unit)

Mount options (1)

- Mount options can be used to control the kernel behavior on the entire filesystem
- **`bsddf`**, **`minixdf`**: changes the behavior of **`statfs()`**
- **`bsdgroups|sysvgroups`**: uses BSD or SVR4 semantics when creating files
  - BSD: gid is inherited from the parent directory
  - SVR4: gid is the process primary gid, or the parent directory's gid if the directory is setgid
- **`check=none|normal|strict`**: controls the level of consistency checks made by the kernel during operations
- **`debug`**: adds some debug messages

Mount options (2)

- **`errors=continue|remount-ro|panic`**: controls the error behavior
- **`resgid=gid`**, **`resuid=uid`**: specifies the group or user, which can use the reserved blocks (“Les options du métro”:-)
- **`sb=block`**: uses an alternate copy of the superblock
- **`sync`**: uses synchronous updates à la BSD:
  - metadata (inodes, bitmaps, directory entries) are written synchronously to the disk
  - poor performances
  - possible corruption in user data may be undetected by fsck

File attributes in Ext2fs

- File attributes can be used on a file basis to modify the kernel behavior
- Append-only: data can only be written to the file in append mode
- Immutable: file cannot be modified
- Secure deletion: the contents of the file is overwritten with random data when the file is deleted
- Synchronous updates: meta-data (inode, indirect blocks) are written synchronously on disk

Tunable parameters

- The superblock contains tunable parameters, used by the kernel code to adapt its behavior
- These parameters can be changed by the **`tune2fs`** utility
- Tunable parameters:
  - mount count: maximal number of mounts before the filesystem is checked
  - check interval: maximal time between two checks
  - reserved blocks: number of reserved blocks for **`root`**
  - error behavior: default value of the **`errors`** mount option
  - default group and user id for the **`resgid`** and **`resuid`** mount option

Fast symbolic links

- Symbolic links usually waste disk space:
  - one inode (128 bytes)
  - one data block (at least 1024 bytes) containing the target name
- Ext2fs implements fast symbolic links:
  - the target name is stored in the inode itself
  - the target name must not contain more than 60 characters
- Advantages:
  - saves one disk block
  - speeds up the symbolic link interpretation (no need to read a data block)

Future Ext2fs development

- Almost no new feature has been added to Ext2fs since Linux 1.0 has been released (only performance enhancements and extension of the limitations)
- It's time to add new features!
- Planned extensions:
  - extension of the attribute management
  - fast files
  - access control lists
  - undelete
  - compression on the fly
  - dump and restore tools

Extension of attributes management

- Currently file attributes are read and written with an **`ioctl()`** call:
  - attributes are only recognized on Ext2 filesystems
  - attributes cannot be used on device files
- New system calls will be added to the kernel:
  - **`fgetattr`**, **`getattr`**: read file attributes
  - **`fsetattr`**, **`setattr`**: write file attributes
- New filesystem operations in the VFS: **`getattr`**, **`setattr`**
- Advantages:
  - uniform way of reading/writing attributes on every filesystem types
  - attributes can be used on every type of file

Fast files

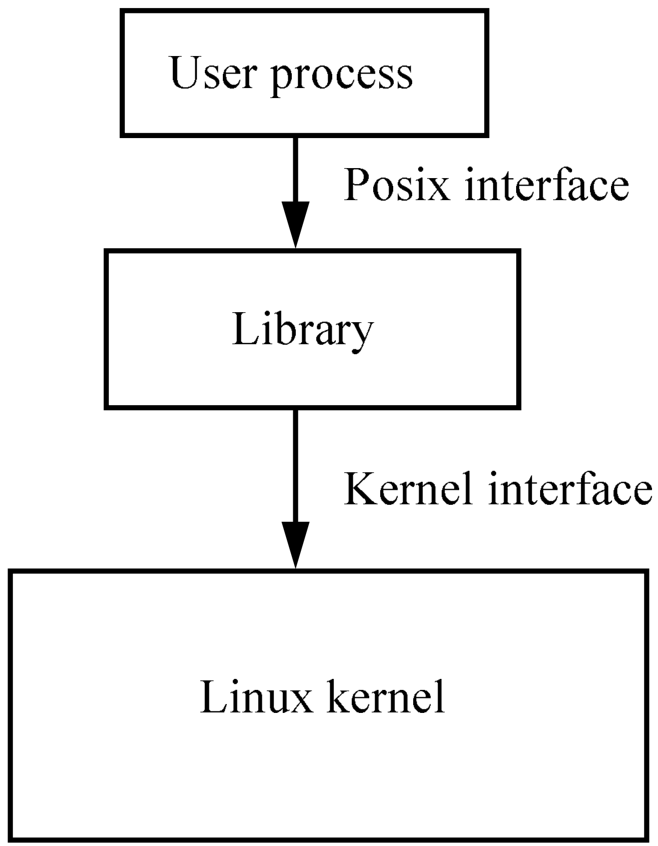
- The fast symbolic link scheme can be extended to files
- Around 5% of files are smaller than 60 characters on a typical Linux root filesytem (including **`/usr`** and **`/var`**)
- Storing the data in the inode itself is easy:
  - the **`ext2_file_read()`** and **`ext2_file_write()`** functions will be modified
  - an **`ext2_mmap()`** function has will be added
- This structural change requires bumping the filesystem revision level
- E2fsck will be modified to take fast files into account

Access Control Lists

- The current file permission scheme is not extensible enough: only 3 file permissions (owner, group, world)
- Access control lists extend this scheme:
  - each entry in the ACL contains a record <user, group, permission>
  - a unlimited number of entries can be associated with any file
- The kernel uses the ACL (if present) to check for permission:
  - checks the owner permission
  - checks the records <user, *, permission>
  - checks the group permission
  - checks the records <*, group, permission>
  - checks the other permission

Implementation of ACL in Ext2fs (1)

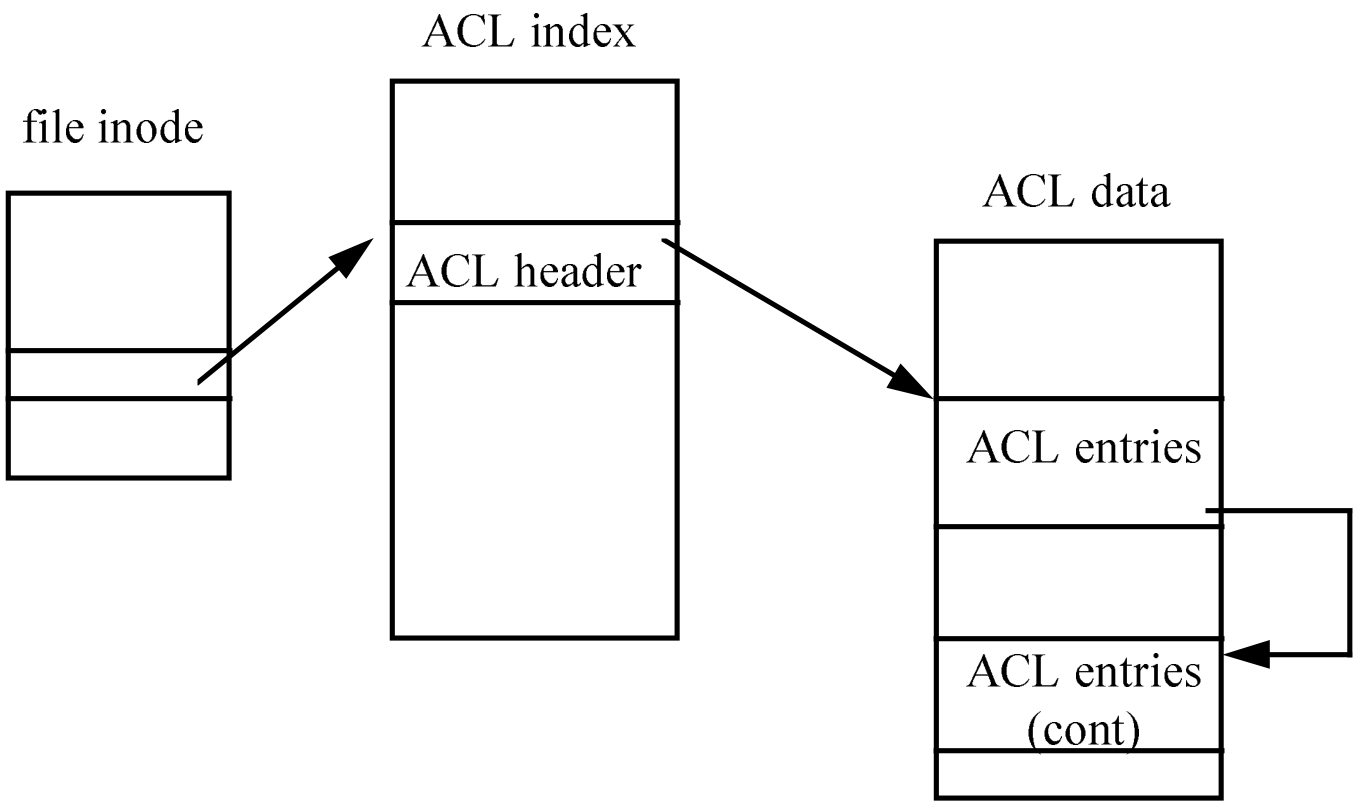
- Three interfaces
- kernel level:
  - very simple interface
  - **`aclctl(filename, operation, arg)`**
  - **`faclctl(filedesc, operation, arg)`**
- programming level:
  - Posix compatible calls
  - handled by a library
- command level: Posix compatible tools



Implementation of ACL in Ext2fs (2)

- The ACL are stored in two reserved inodes:
  - the ACL index inode contains a list of ACL headers
  - the ACL data inode contains the entries (an ACL may be fragmented)
- Each inode contains two pointers to the ACL index inode:
  - default ACL (inherited from the parent directory)
  - file ACL (can be changed by the owner)
- Kernel ACL operations:
  - get the size of an ACL
  - read an ACL
  - write an ACL
  - delete an ACL

Implementation of ACL in Ext2fs (3)



Undelete

- Unix filesystems lack an undelete feature, recovering a deleted file is painfull even for a filesystem guru
- A file attribute requesting undelete already exists but is not honored by the kernel yet
- When an “undeletable” file is removed:
  - the inode and the data blocks are not deallocated
  - the directory entry is moved to a special invisible directory, which holds removed files
- New **`ioctl()`** calls and new tools to act on deleted files:
  - restore a deleted file
  - really remove a deleted file

Compression

- Goals:
  - automatic compression on **`write()`**
  - automatic decompression on **`read()`**
  - easy integration of compression algorithms
  - flexible configuration
  - security
- Existing packages:
  - DouBle (Jean-Marc Verbavatz)
  - E2compress (Antoine Dumesnil de Maricourt)
  - HUFS (Michael Beck)
- A merge of E2compress and HUFS will probably be integrated in Ext2fs very soon

Dump and restore

- The 4.4BSD dump and restore tools have been ported to Ext2fs by using the Ext2fs library
- Private test version distributed 9 months ago:
  - no major bugs (only small ones)
  - but, the dump format is not compatible with BSD based operating systems
- Public release is due in june:
  - bugs have been fixed
  - the dump format is now compatible with BSD OS (including commercial ones, e.g. SunOS 4)
  - new features have been added (e.g. it is possible to backup subtrees)

Conclusion

- The Ext2fs is probably the most widely used file system in the Linux community
- It provides:
  - Unix features
  - advanced features
  - excellent performances
- It has provision for future extensions
- It has been ported to other operating systems:
  - GNU Hurd
  - VSTa

Future Development

- New kernel features:
  - Fast files
  - Access Control Lists
  - Undelete
  - File Compression
- New tools: dump and restore backup suite
- New ports:
  - GNU Hurd
  - LITES
  - FreeBSD