CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

Directory Structure

Directory basically just contains a list of (entry name, inode number) pairs.
For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory.
For each string, there may also be a length (assuming variable-sized names)

inum	reclen	strlen	name
5	12	2	.
2	12	3	..
12	12	4	foo
13	12	4	bar
24	36	28	foobar_is_a_pretty_longname

Each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry.

Directory Structure

•Directory is a special type of file and has inode and data blocks (which store the file records)

• In the inode table, an inode entry with the type field of the inode marked as “directory” instead of “regular file”)

•The directory has data blocks pointed to by the inode (and perhaps, indirect blocks);

• These data blocks live in the data block region of the file system. Thus, on-disk structure remains unchanged.

Free Space Management

How to track free blocks?

–Bitmaps, for inodes and data blocks, store one bit per block to indicate if free or not [If disk is nearly full this becomes very expensive and doesn't produce much locality] - Solution ?

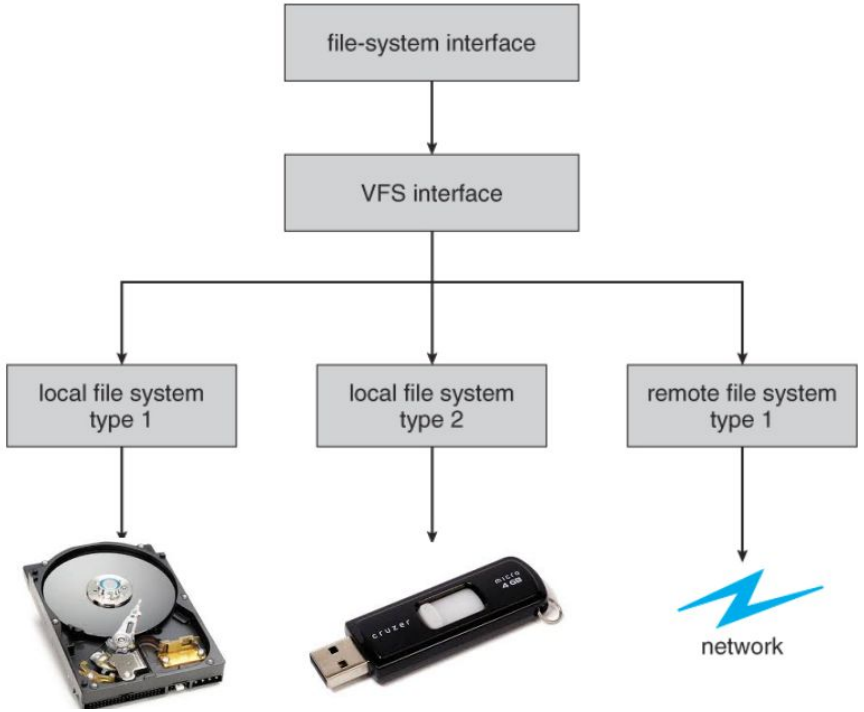
–Free list, super block stores pointer to first free block, a free block stores address of next block on list [Linked list of free blocks]

–More complex structures can also be used

•File systems may use heuristics–eg. A group of closely spaced free blocks

Virtual File System

- How do we seamlessly support multiple file systems and devices?



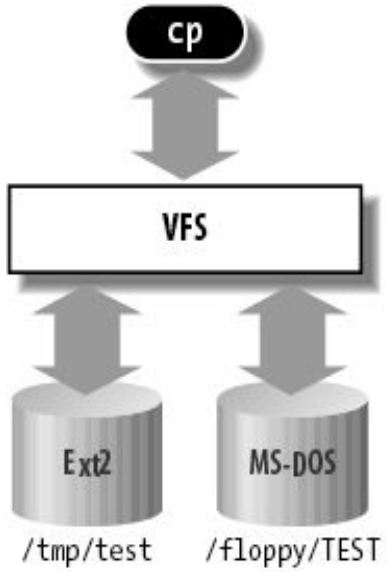
What if files are more than on one device ?

What if device(s) work better with different FS implementation ?

What if files are not on a local device but accessed via a network ?

VFS

A kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.



(a)

```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(b)

cp /floppy/TEST /tmp/test
/floppy is the mount point of MS-DOS disk and */tmp* is a normal Second Extended Filesystem (Ext2) directory

VFS is an abstraction layer between the application program and the filesystem implementations
cp program is not required to know the filesystem types of */floppy/TEST* and */tmp/test*

cp interacts with the VFS by means of generic system calls

VFS

File systems differ in implementations of data structures (e.g., organization of file records in directory)

- Linux supports virtual file system (VFS) abstraction
- VFS looks at a file system as objects (files, directories, inodes, superblock) and operations on these objects(e.g., lookup filename in directory)
- System call logic is written on VFS objects
- To develop a new file system, simply implement functions on VFS objects and provide pointers to these functions to kernel
- Syscall implementation does not have to change with file system implementation details

Acess Methods [Read, Write]

Let us assume that the file system has been mounted and thus that the superblock is already in memory. Everything else (i.e., inodes, directories) is still on the disk.
Read – Suppose you want to simply open a file (e.g.,/foo/bar), read it, and then close it.
let's assume the file is just 12KB in size (i.e., 3 blocks)
1. issue an open("/foo/bar", O_RDONLY) call, the file system first needs to find the inode for the file bar, to obtain some basic information about the file
Why open? To have the inode readily available (in memory) for future operations on file
–Open returns fd which points to in-memory inode
–Reads and writes can access file data from inode

2. File system must be able to find the inode, but all it has right now is the full pathname. The file system must traverse the pathname and thus locate the desired inode.
The pathname of the file is traversed, starting at root
–Inode of root is known, to boot strap the traversal [will read from disk the inode of the root directory] [use these on-disk pointers to read through the directory, in this case looking for an entry for **foo]**.
–Recursively do: fetch inode of parent directory, read its data blocks, get inode number of child, fetch inode of child. Repeat until end of path
[FS reads the block containing the inode of foo and then its directory data, finally finding the inode number of bar.]
If new file, new inode and data blocks will have to be allocated using bitmap, and directory entry updated

The final step of open()is to read bar’s inode into memory;
the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.
Once open, the program can then issue a read() system call to read from the file.
The first read (at offset 0 unless lseek()has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block;
it may also update the inode with a new last-accessed time.
The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**			**read**				
				read			**read**			
					read					
read()					**read**					
					write			**read**		
read()					**read**					
					write				**read**	
read()					**read**					
					write					**read**

• Global open file table –One entry for every file opened (even sockets, pipes)–Entry points to in-memory copy of inode(other data structures for sockets and pipes)
•Per-process open file table
–Array of files opened by a process
–File descriptor number is index into this array–Per-process table entry points to global open file table entry
–Every process has three files (standard in/out/err)open by default (fd 0, 1, 2)
•Open system call creates entries in both tables and returns fd number

Writing to a disk

1. The file must be opened (as earlier).
2. Then, the application can issue write()calls to update the file with new contents.
3. Finally, the file is closed
4. Writing to the file may also allocate a block (unless the block is being overwritten).
5. When writing out a new file, each write not only has to write data to disk but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode).

Write

Each write to a file logically generates five I/Os:

- one to read the data bitmap (which is then updated to mark the newly-allocated block as used)
- one to write the bitmap (to reflect its new state to disk),
- two more to read and then write the inode (which is updated with the new block’s location)
- Finally one to write the actual block itself

For reading/writing file
–Access in-memory inode via file descriptor
–Find location of data block at current read/write offset
–Fetch block from disk and perform operation
–Writes may need to allocate new blocks from disk using bitmap of free blocks
–Update time of access and other metadata in inode

File creation

To create a file, the FS must not only allocate an inode, but also allocate space within the directory containing the new file. T
Total amount of I/O traffic to do so is quite high:
one read to the inode bitmap (to find a free inode)
one write to the inode bitmap (to mark it allocated)
one write to the new inode itself (to initialize it)
one to the data of the directory (to link the high-level name of the file to its inode number)
one read and write to the directory inode to update it.
If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		read write	read	read write	read write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

You can see how much work it is to create the file:
10 I/Os in this case, to walk the path name and then finally create the file.
You can also see that each allocating write costs 5 I/Os: a pair to read and update the inode, another pair to read and update the data bitmap, and then finally the write of the data itself.
How can a file system accomplish any of this with reasonable efficiency?

Disk Buffer Cache

Use part of main memory to retain recently-accessed disk blocks.
Blocks that are referenced frequently (e.g., indirect blocks for large files) are usually in the cache.
This solves the problem of slow access to large files.
Originally, buffer caches were fixed size.
As memories have gotten larger, so have buffer caches.

Disk buffer cache

- Results of recently fetched disk blocks are cached
  - LRU to evict if cache is full
- File system issues block read/write requests to block numbers via buffer cache
  - If block in cache, served from cache, no disk I/O
  - If cache miss, block fetched to cache and returned to file system
- Writes are applied to cache block first
  - Synchronous/write-through cache writes to disk immediately
  - Asynchronous/write-back cache stores dirty block in memory and writes back after a delay

Synchronous writes: immediately write through to disk.

- Safe: data won't be lost if the machine crashes
- Slow: process can't continue until disk I/O completes
- May be unnecessary:
  - Many small writes to the same block
  - File deleted soon (e.g., temporary files)

Delayed writes: don't immediately write to disk:

- Wait a while (30 seconds?) in case there are more writes to a block or the block is deleted
- Fast: writes return immediately
- Dangerous: may lose data after a system crash

- Unified page cache in OS –Free pages allocated to both processes and disk buffer cache from common pool
- Two benefits

–Improved performance due to reduced disk I/O(one disk access for multiple reads and writes)

–Single copy of block in memory (no inconsistency across processes)

- Some applications like databases may avoid caching altogether, to avoid inconsistencies due to crashes: direct I/O

Run Ahead

FS predicts that the process will request next block
FS goes ahead and requests it from the disk ..while the process is computing on previous block!
–When the process requests block, it will be in cache
–Compliments the disk cache, which also is doing read ahead
For sequentially accessed files can be a big win
Unless blocks for the file are scattered across the disk and File systems try to prevent that, though (during allocation)

Caching Writes

Applications assume writes make it to disk -- As a result, writes are often slow even with caching
Write-behind» Maintain a queue of uncommitted blocks » Periodically flush the queue to disk » Unreliable
"Battery backed-up RAM (NVRAM)" » As with write-behind, but maintain queue in NVRAM» Expensive
Log-structured file system » Always write next block after last block written » Complicated

Problem with failures

Crash failures can occur at any moment (eg, power outage).

- Data saved on disk should still be available on restart after a crash. Our file system may be impacted by such a crash!
- A crash may leave the file system in an inconsistent state

Update operations on the file system (create dir, create file, writefile) require several I/O operations.

- What if a crash occurs before all operations related to an update are completed?

The file system will be in an inconsistent state

Append one data block to a file: requires 3 writes (data bitmap, the file inode, the data block)
Only data block is written: FS remains consistent, data is lost
Only inode is written: Inode points to trash, bitmap and inode are not consistent
Only bitmap is written: A data block is lost (space leak)
Adding block to file: free list was updated to indicate block in use, but file descriptor wasn't yet written to point to block.
Creating link to a file: new directory entry refers to file descriptor, but reference count wasn't updated in file descriptor.

Solutions

Ideal solution •Make all updates in one atomic step to avoid any inconsistencies
Impossible, the disk does one write at a time
2 existing techniques •File system checker (fsck) •Journaling
Fsck - Basic idea
•Let inconsistencies happen and try to fix them on restart
•Scan the file system (superblock, bitmaps, inodes) and check for inconsistencies
•Extremely inefficient! -- Checking the whole FS when maybe a single inode is inconsistent.

Unix fsck ("file system check")

During every system boot fsck is executed.
Checks to see if disk was shut down cleanly; if so, no more work to do.
If disk didn't shut down cleanly (e.g., system crash, power failure, etc.), then scan disk contents, identify inconsistencies, repair them.
Example: block in file and also in free list
Example: reference count for a file descriptor doesn't match the number of links in directories
Example: block in two different files
Example: file descriptor has a reference count > 0 but is not referenced in any directory.

Limitations of fsck

Restores disk to consistency, but doesn't prevent loss of information; system could end up unusable.
Security issues: a block could migrate from the password file to some other random file.
Can take a long time: 1.5 hours to read every block in a medium-size disk today.
Can't restart system until fsck completes. As disks get larger, recovery time increases.

Ordered Writes

Prevent certain kinds of inconsistencies by making updates in a particular order.

- For example, when adding a block to a file, first write back the free list so that it no longer contains the file's new block.
- Then write the file descriptor, referring to the new block.
- What can we say about the system state after a crash?
- In general:
  - Never write a pointer before initializing the block it points to (e.g., indirect block).
  - Never reuse a resource before nullifying all pointers to it.
  - Never clear last pointer to a live resource before setting new pointer (e.g. mv).

Result: no need to wait for fsck when rebooting

- Run fsck in background to reclaim lost storage.

Problem:

- Requires lots of synchronous metadata writes, which slows down file operations.

Improvement:

- Don't actually write the blocks synchronously, but record dependencies in the buffer cache.
- For example, after adding a block to a file add dependency between file descriptor block and free list block.
  - When it's time to write the file descriptor back to disk, make sure that the free list block has been written first.
- Tricky to get right: potentially end up with circular dependencies between blocks.

Journaling

- Write-ahead logging (database community) -- When updating the disk, before over-writing the structures in place, first write down a little note (some where else on the disk, in a well-known location)
- If a failure occurs in the middle of the update, we can read the journal on restart and try again (or at least fix inconsistencies).
- Solution used by many FS including Linux ext3, Linux ext4and Windows NTFS.
- Linux ext3 looks the same as ext2 except that a journal is added to the file system (one more region

Example: adding a block to a file

Log entry: "I'm about to add block 99421 to file descriptor 862 at block index 93"
Then the actual block updates can be carried out later.
If a crash occurs, replay the log to make sure all updates are completed on disk.
Problem: log grows over time, so recovery could be slow.
Solution #1: checkpoint
Occasionally stop and flush all dirty blocks to disk.
Once this is done, the log can be cleared.
Solution #2: keep track of which parts of the log correspond to which unwritten blocks; as blocks get written to disk, can gradually delete old portions of the log that are no longer needed.
Typically the log is used only for metadata (free list, file descriptors, indirect blocks), not for actual file data.

Remaining Problems

Can still lose recently-written data after crash

- Solution: apps can use fsync to force data to disk.

Disks fail

- One of the greatest causes of problems in large datacenters
- Solution: replication or backup copies

Disk writes are not atomic:

- If a block is being written at the time of the crash, it may be left in inconsistent state (neither old contents nor new).
- At the level of sectors, inconsistencies are detectable; after crash, sector will be either
  - Old contents
  - New contents
  - Unreadable trash

- But, blocks are typically multiple sectors. After crash:
  - Sectors 0-5 of block may have new contents.
  - Sectors 6-7 of block may have old contents.
- Example: appending to log
  - If adding new log entries to an existing log block, crash could cause old info in the block to be lost.
- Solution:
  - Replicate log writes (if crash corrupts one of the logs, the other will still be safe).
  - Add checksums and/or versions to detect incomplete writes.

Conclusions:

- Must decide what kinds of failures you want to recover from.
- To get highest performance, must give up some crash recovery capability.