11: File System Basics

Last Modified:
10/25/2002 9:40:49 AM

File System Basics

- FS are probably the OS abstraction that average user is most familiar with
  - Files
  - Directories
  - Access controls (owners, groups, permissions)

Files

- A file is a collection of data with system maintained properties like
  - Owner, size, name, last read/write time, etc.
- Files often have "types" which allow users and applications to recognize their intended use
- Some file types are understood by the file system (mount point, symbolic link, directory)
- Some file types are understood by applications and users (.txt, .jpg, .html, .doc, ...)
  - Could the system understand these types and customize its handling?

Basic File Operations

UNIX	Windows
create (name)	CreateFile(name, CREATE)
open (name, mode)	CreateFile(name, OPEN)
read (fd)	ReadFile(handle)
write(fd)	WriteFile (handle)
sync(fd)	FlushFileBuffers(handle)
seek(fd, pos)	SetFilePointer(handle)
close(fd)	CloseHandle(handl)
unlink (name)	DeleteFile(name)
rename (old, new)	CopyFile(name)
	MoveFile(name)

Directories

- Directories provide a way for users to organize their files *and* a convenient way for users to identify and share data
- Logically directories store information like file name, size, modification time etc (Not always kept in the directory though..)
- Most file systems support hierarchical directories (/usr/local/bin or C:\WINNT)
  - People like to organize information hierarchically
- Recall: OS often records a current working directory for each process
  - Can therefore refer to files by absolute and relative names

Directories are special files

- Directories are files containing information to be interpreted by the file system itself
  - List of files and other directories contained in this directory
  - Some attributes of each child including where to find it!!
- How should the list of children be organized?
  - Flat file?
  - B-tree?
- Many systems have no particular order, but this is extremely bad for large directories!

Multiple parent directories?

- One natural question is "can a file be in more than one directory"?
- Soft links
  - Special file interpreted by the FS (like directories in that sense)
  - Tell FS to look at a different pathname for this file
  - If file deleted or moved, soft link will point to wrong place
- Hard links
  - Along with other file info maintain reference count
  - Delete file = decrement reference count
  - Only reclaim storage when reference count does to 0

Path Name Translation

- To find file "/foo/bar/baz"
  - Find the special root directory file (how does FS know where that is?)
  - In special root directory file, look for entry foo and that entry will tell you where foo is
  - Read special directory file foo and look for entry bar to tell you where bar is
  - Find special directory file bar and look for entry baz to tell you where baz is
  - Finally find baz
- FS can cache common prefixes for efficiency

File Buffer Cache

- Cache Data Read
  - Exploit temporal locality of access by caching pathname translation information
  - Exploit temporal locality of access by leaving recently accesses chunks of a file in memory in hopes that they will be accessed again (let app give hint if not?)
  - Exploit spatial locality of access by bringing in large chunks of a file at once
- Data written is also cached
  - For correctness should be write-through to disk
  - Normally is write-behind
    - FS periodically walks the buffer cache and "flushes" things older than 30 seconds to disk
    - Unreliable!
- Usually LRU replacement

File Buffer Cache

- Typically cache is system wide (shared by all processes)
  - Shared libraries and executables and other commonly accessed files likely to be in memory already
- Competes with virtual memory system for physical memory
  - Processes have less memory available to them to store code and data (address space)
  - Some systems have integrated VM/FS caches

Protection System

- Most FS implement a protection scheme to control:
  - Who can access a file
  - How they can access it (e.g. read/write/exec/..)
- Any protection system dictates whether a given action performed by a given subject on a given object should be allowed. In this case:
  - Objects = files
  - Principles = users
  - Actions = operations
- We'll talk more about protection systems later in the semester

File Systems

- We talked a bit about disk internals
- Despite complex internals, disks export a simple array of sectors
- How do we go from that to a file system?

Exercise for the Reader ☺

- If you were going to build your own file system on top of a fixed sized file what would you do?
  - What other information would you need to store there besides file data and directory data?
  - How would you organize things?

Some questions

- Would you keep each file together sequentially?
  - If you did, what would you do if a file grew or shrunk?
  - If not, how would you keep track of the multiple pieces?

File Layout

- Option 1: All blocks in a file must be allocated contiguously
  - Only need to list start and length in directory
  - Causes fragmentation of free space
  - Also causes copying as files grow
- Option 2: Allow files to be broken into pieces
  - Fixed sized pieces (blocks) or variable sized pieces (extents)?
  - If we are going to allow files to be broken into multiple pieces how will we keep track of them ?

Blocks or Extents?

- If fixed sized block then store just starting location for each one
- If variable sized extent need to store starting location and length
  - But maybe you can have fewer extents?
- Blocks = less external fragmentation
- Extents = less internal fragmentation

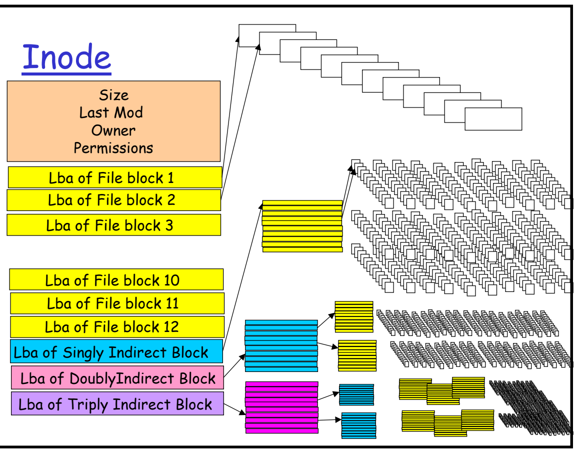
Finding all Parts of a File

- Option 2A: List all blocks in the directory
  - Directories will get pretty big and also must change the directory everytime a file grows or shrinks
- Option 2B: Linked structure
  - Directory points to first piece (block or extent), first one points to second one
  - File can expand and contract without copying
  - Good for sequential access, terrible for other kinds
- Option 2C: Indexed structure
  - Directory points to index block which contains pointers to data blocks
  - Good for random access as well as sequential access

Unix Inodes

- Inode = index nodes
  - Files broken into fixed size blocks
  - Inodes contain pointers to all the files blocks
  - Directory points to location of inodes
- Each inode contains 15 block pointers
  - First 12 point directly to data blocks
  - Then single, doubly and triply indirect blocks
- Inodes often contain information like last modification time, size, etc. that could logically be associated with a directory
- Note: Indirect blocks sometime numbered as file blocks -1, -2, etc.

Inode



Max file size?

- Assume:
  - 4K data pages and indirect blocks
  - Lbas are typically 4 bytes
- First 48K directly reachable from inodre
- One singly indirect block reaches 1024 more blocks = 4M
- One double indirect blocks points to 1024 more singly indirect blocks which each point to 4 M of data = 4 GB
- One triply indirect block points to 1024 more doubly indirect blocks which each point to 4 GB of data = 4 TB
- Max file or directory size = 4TB + 4GB + 4 MB + 48 K

Other index structures?

- Why this particular index structure?
  - ?
  - Direct pointers to first 12 blocks is good for small files
- Could you imagine other index structures?
  - Definitely
  - Flat vs Multilevel Index structures?

Path Name Traversal Revisited

- Directories are just special files so they have inodes of their own
- To find "/foo/bar/baz" (assuming nothing is cached)
- Look in super block and find location of I-node for /
- Read inode for /, find location of first data block of /
- Read first data block of /
- Repeat with all blocks of / until find entry for foo if read until block 13 then must read singly indirect block first…
- When find entry for foo gives address of I-node for foo, read inode for foo,..

More questions

- Remember you are building your own FS
- When someone creates a new file where would you put it?
  - How would you find free space?
  - How much space would you look for (how do you know how big the file will get?)
  - Would you take the first acceptable chunk of free space you found?
- If someone deleted a file what would happen?

Keeping track of free space

- Linked list of free space
  - Just put freed blocks on the end and pull blocks from front to allocate
  - Hard to manage spatial locality (why important?)
  - If middle of list gets corrupted how to repair?
- Bit map
  - Divide all space into blocks
  - Bit per block (0 = free; 1 = allocated)
  - Easy to find groups of nearby blocks
  - Useful for disk recovery
  - How big? If had 40 GB disk, then have 10M of 4K blocks is each needs 1 bit then 10M/8 = 1.2 MB for the bit map

Answers?

- We are going to look at two different file systems
  - Fast File System (FFS)
  - Log-Structured File Systems (LFS)
- Remember *your* answers to the questions we just posed, at the end of today, if you think your answers are better then maybe you will go on to write your own file system (MeFS)

Right answer?

- Remember much like CPU or disk scheduling algorithms, "right" answer depends a lot on your workload
- Are there many small files (< 48 K) or all big files?
- Are files usually read sequentially from the beginning or randomly?
- Are files read and written equally?
- Are files in the same directory usually accessed together?
- How to find out?
  - Analyze static FS snapshots
  - Take FS access traces
  - Even then which systems do you look at? Are they "representative"?

Outtakes

Some questions

- What would your directory structure look like (directory_t ?)
- How would you find the root directory? What if the root directory got really really big?

File Interfaces

- Structured Record files
  - Think of file as containing data structures not bytes
- Record-Stream Translation
  - When access record, read/write the correct number of bytes
- Byte Stream Files
  - Appear to user as stream of bytes
- Stream-Block Translation
  - If read/write single byte from block get the whole thing
- Raw Storage Interface
  - Sectors or FS blocks

Byte Stream File Interface

- fileId = open(fileName)
- Close (fileID)
- Seek(fileId, filePosition)
- Read (fileId, buffer, length)
- Write (fileId, buffer, length)

Record Oriented File Interface

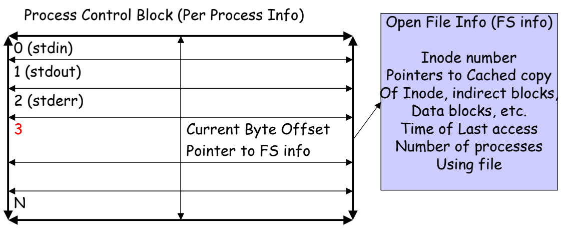
- fileId = open(fileName)
- Close (fileId)
- getRecord (fileId, record)
- putRecord (fileId, record)
- Seek(fileId, recordNum)

Open system call

- fileId = Open (fileName)
- Read directory containing base file name
- Get Inode of file
- Determine permission to open file (different modes: read, read/write, append, etc.)
- OS creates an internal file descriptor for this file or may find that one already exists if other processes are accessing this file
- Allocate resources (buffers etc.) to support access to the file
- Create an entry in the process control block for this file; Process control block has an array of file information
- Return the index into the array in the PCB as the fileId

Accessing an open file

```
fileId = open ("/foo/bar/baz", flags);
read(fileId, buffer, len);
        syscall(READ, fileId, buffer,len);
```



- Other kinds of "special files"
- Device special files