Implementing File Systems

- Basic file system structure
- Allocating disk blocks to files
  - » Deciding which blocks to include in a file
  - » Keeping track of the blocks in a file
- Managing free space on the disk
- Implementing directories
  - » Translating a human-readable name to a file identifier
  - » Keeping track of file metadata
- Improving file system efficiency & performance
  - » Using less disk space
  - » Making the file system faster
- Recovery: when file systems go bad

Basic File System Structure

- A file is
  - » A logical storage unit, tracked as a whole by the file system
  - » A collection of logically related information
- File system resides on
  - » Disks (usually)
  - » Tape (occasionally)
  - » Memory (RAM disk, flash memory)
- File system organized into logical layers: allows different file systems to share code
- File metadata is stored in a <u>file control block</u> (called an <u>inode</u> in Unix)

OS interface

↓

Directory routines:
lookup file, create file, etc.

↓

File organization routines:
read & write file blocks

↓

Low-level routines:
allocate & free disk blocks

↓

Device drivers

↓

Hardware (disks, etc.)

Using a File System: The OS View

- Opening a file
  - » Find the file and check to ensure that the user is allowed to perform the desired operation (read, write, etc.)
  - » Allocate an entry in the “open file” table and return a “handle” to it to the user (file descriptor)
  - » Process-only open file table vs. global open file table
- Closing a file
  - » Write out all changes to the metadata
  - » Deallocate the file control block in memory
- Mounting a file system: make a file system available to users
  - » Identify the file system’s position in the directory structure
  - » Locate the directory information on the disk
  - » Build structures in memory that allow the OS to use the file system

Allocating Blocks to Files

- Files contain data stored in many file blocks
  - » File block is minimum unit of disk space allocation in file system
  - » File block size may be larger than disk block size
- Goal: keep track of which blocks contain the data in this file
  - » Allow both sequential and random access efficiently
  - » Use as little space as possible
  - » Allow files to grow (shrink not necessary, only truncate)
- Allocation decisions require
  - » How are blocks on disk grouped?
  - » How can the file system figure out which disk block corresponds to a particular file block?
- For all these examples, assume file blocks are 1024 bytes

Contiguous Allocation

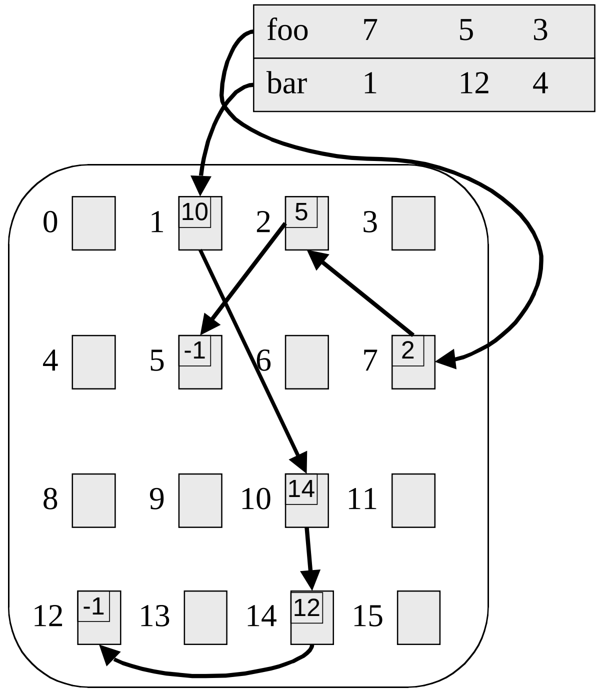
- Data in each file is stored in consecutive blocks on disk
- Simple & efficient indexing
  - » Starting location (block #) on disk (`start`)
  - » Length of the file in blocks (`length`)
- Random access well-supported
- Difficult to grow files
  - » Must pre-allocate all needed space
  - » Wasteful of storage if file isn't using all of the space
- Logical to physical mapping is easy

```
blocknum = (pos / 1024) + start;
offset_in_block = pos % 1024;
```


Linked Allocation

- File is a linked list of disk blocks
  - » Blocks may be scattered around the disk drive
  - » Block contains both pointer to next block and data
  - » Files may be as long as needed
- New blocks are allocated as needed
  - » Linked into list of blocks in file
  - » Removed from list (bitmap) of free blocks

name	start	end	size
foo	7	5	3
bar	1	12	4



Finding Blocks: Linked Allocation

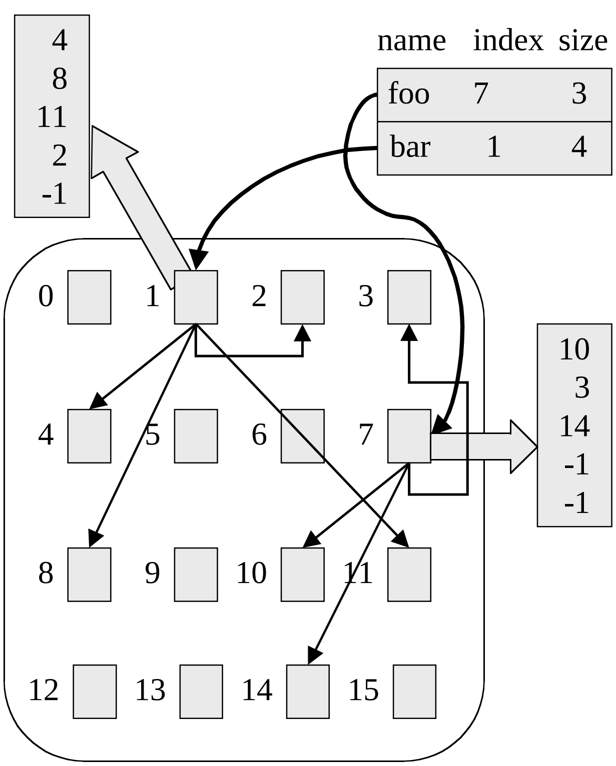
- Directory structure is simple
  - » Starting address looked up from directory
  - » Directory only keeps track of first block (not others)
- No wasted space - all blocks can be used
- Random access is difficult: must always start at first block!
- Logical to physical mapping is done by

```
block = start;
offset_in_block = pos % 1020;
for (j = 0; j < pos / 1020; j++) {
  block = block->next;
}
```

  - » Assumes that “next” pointer is stored at end of block
  - » May require a long time for seek to random location in file

Using a Block Index for Allocation

- Store file block addresses in an array
  - » Array itself is stored in a disk block
  - » Directory has a pointer to this disk block
  - » Non-existent blocks indicated by -1
- Random access easy
- Limit on file size?



Finding Blocks with Indexed Allocation

- Need location of index table: look up in directory
- Random & sequential access both well-supported: look up block number in index table
- Space utilization is good
  - » No wasted disk blocks (allocate individually)
  - » Files can grow and shrink easily
  - » Overhead of a single disk block per file
- Logical to physical mapping is done by
```
block = index[block % 1024];
offset_in_block = pos % 1024;
```
- Limited file size: 256 pointers per index block, 1 KB per file block -> 256 KB per file limit

Larger Files with Indexed Allocation

- How can indexed allocation allow files larger than a single index block?
- Linked index blocks: similar to linked file blocks, but using index blocks instead
- Logical to physical mapping is done by
```
index = start;
blocknum = pos / 1024;
for (j = 0; j < blocknum /255); j++) {
  index = index->next;
}
block = index[blocknum % 255];
offset_in_block = pos % 1024;
```
- File size is now unlimited
- Random access slow, but only for very large files

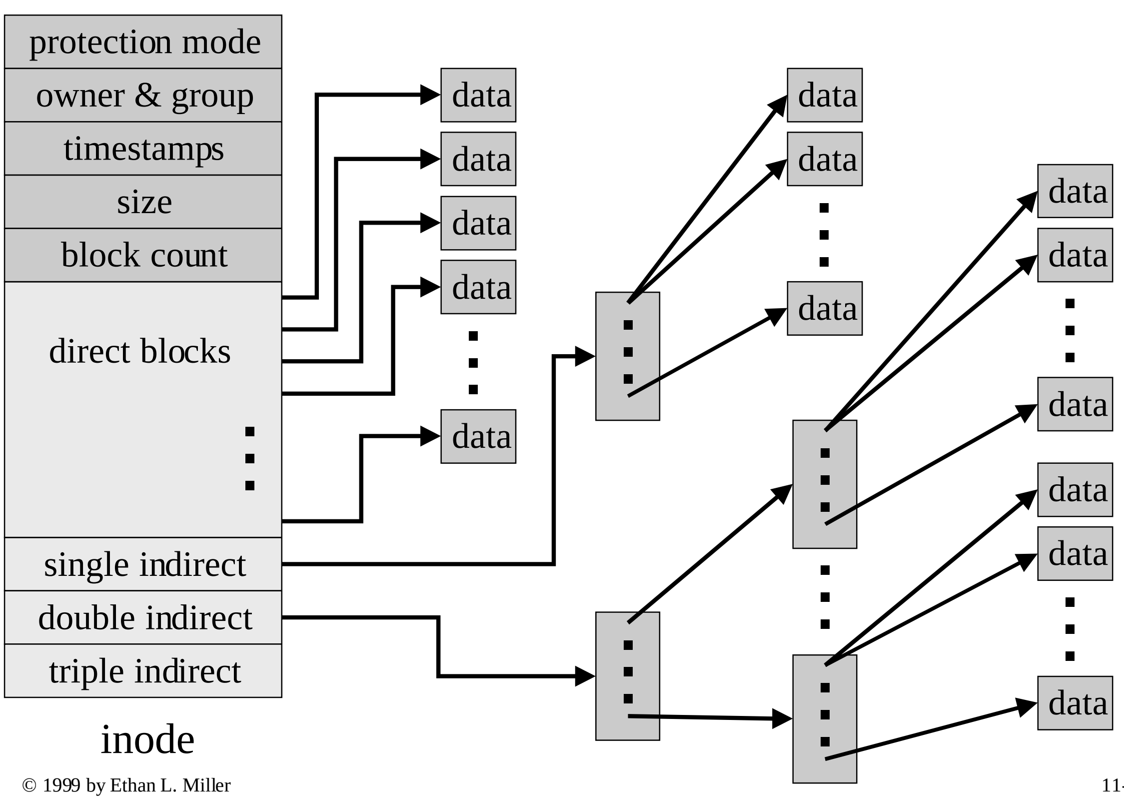
Two-Level Indexed Allocation

- Allow larger files by creating an index of index blocks
  - » File size still limited, but much larger
  - » Limit for 1 KB blocks = 1 KB * 256 * 256 = 2^{26} bytes = 64 MB
- Logical to physical mapping is done by

```
blocknum = pos / 1024;
index = start[blocknum / 256)];
block = index[blocknum % 256]
offset_in_block = pos % 1024;
```

  - » `Start` is the only pointer kept in the directory
  - » Overhead is now at least two blocks per file
- This can be extended to more than two levels if larger files are needed...

Unix FFS Allocation Scheme



inode

More on Unix FFS

- First few block pointers kept in directory
  - » Small files have no extra overhead for index blocks
  - » Reading & writing small files is very fast!
- Indirect structures only allocated if needed
- For 4 KB file blocks (common in Unix), max file sizes are:
  - » 48 KB in directory (usually 12 direct blocks)
  - » 1024 * 4 KB = 4 MB of additional file data for single indirect
  - » 1024 * 1024 * 4 KB = 4 GB of additional file data for double indirect
  - » 1024 * 1024 * 1024 * 4 KB = 4 TB for triple indirect
- Maximum of 5 accesses for any file block on disk
  - » 1 access to read inode & 1 to read file block
  - » Maximum of 3 accesses to index blocks
  - » Usually much fewer (1-2) because inode in memory

Block Allocation with Extents

- Reduce space consumed by index pointers
  - » Often, consecutive blocks in file are sequential on disk
  - » Store <block,count> instead of just <block> in index
  - » At each level, keep total count for the index for efficiency
- Lookup procedure is:
  - » Find correct index block by checking the starting file offset for each index block
  - » Find correct <block,count> entry by running through index block, keeping track of how far into file the entry is
  - » Find correct block in <block,count> pair
- More efficient if file blocks tend to be consecutive on disk
  - » Allocating blocks like this allows faster reads & writes
  - » Lookup is somewhat more complex

Managing Free Space: Bit Vector

- Keep a bit vector, with one entry per file block
  - » Number bits from 0 through n-1, where n is the number of file blocks on the disk
  - » If bit[j] == 0, block j is free
  - » If bit[j] == 1, block j is in use by a file (for data or index)
- If words are 32 bits long, calculate appropriate bit by:

```
wordnum = block / 32;
bitnum = block % 32;
```

- Search for free blocks by looking for words with bits unset (words != 0xffffffff)
- Easy to find consecutive blocks for a single file
- Bit map must be stored on disk, and consumes space
  - » Assume 4 KB blocks, 8 GB disk => 2M blocks
  - » 2M bits = 2^{21} bits = 2^{18} bytes = 256KB overhead

Managing Free Space: Linked List

- Use a linked list to manage free blocks
  - » Similar to linked list for file allocation
  - » No wasted space for bitmap
  - » No need for random access unless we want to find consecutive blocks for a single file
- Difficult to know how many blocks are free unless it's tracked elsewhere in the file system
- Difficult to group nearby blocks together if they're freed at different times
  - » Less efficient allocation of blocks to files
  - » Files read & written more because consecutive blocks not nearby

Issues with Free Space Management

- OS must protect data structures used for free space management
- OS must keep in-memory and on-disk structures consistent
  - » Update free list when block is removed: change a pointer in the previous block in the free list
  - » Update bit map when block is allocated
    - – Caution: on-disk map must never indicate that a block is free when it's part of a file
    - – Solution: set bit[*j*] in free map to 1 on disk *before* using block[*j*] in a file and setting bit[*j*] to 1 in memory
    - – New problem: OS crash may leave bit[*j*] == 1 when block isn't actually used in a file
    - – New solution: OS checks the file system when it boots up…
- Managing free space is a big source of slowdown in file systems

Implementing Directories

- Two types of information
  - » File names
  - » File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - » Linear list of files (often itself stored in a file)
    - – Simple to program
    - – Slow to run
  - » Hash table: name hashed and looked up in file
    - – Decreases search time: no linear searches!
    - – May be difficult to expand
    - – Can result in collisions (two files hash to same location)
  - » Tree structure
    - – Either of above choices in a tree structure
    - – Natural choice for graph-based directories (like Unix)

Directory Structures in Unix

- Information stored in two places
  - » File metadata stored in inodes
  - » File names stored in directories (special kind of file)
- Information in directories
  - » File name
  - » Inode number (used to look up metadata to find file data)
  - » Pointers to subdirectories look the same as files!
- Inodes
  - » Stored in arrays spread throughout the disk (cylinder groups)
  - » Indexed linearly by inode number: file system can quickly locate an inode if its number is known
  - » Limited to a certain number, determined when the file system is put onto the disk (make sure there are enough!)

File System Performance

- Many factors determine file system performance
  - » Disk allocation algorithms
  - » Directory management
    - – Location of directories
    - – Type of information stored in directories
- Performance can be improved by
  - » File system <u>cache</u>: store frequently used information (directory & file data) in main memory instead of going to disk each time
  - » <u>Read-ahead</u>: read blocks past current read point without being explicitly asked, and cache them in memory for later use
  - » <u>Delayed write</u>: hold written blocks in memory rather than writing them immediately to disk
    - – Blocks may change again before being written
    - – Files may be deleted before they're actually written
    - – Caution: more exposure to loss of data from OS crash

Improving Unix FS Performance

- Cache commonly used blocks in main memory
  - » File data blocks
  - » Inode information for both open and recently open files
- Delay writes to disk by up to 30 seconds
  - » Many files are deleted before then (e.g., compiler temporaries)
  - » Other files have several writes within that time
- Read one block ahead of current request
  - » Block may be read into memory before next request arrives
  - » Subsequent request may be satisfied immediately
  - » May increase disk utilization (some reads go unused)
- Allocate file data blocks near the file's inode
  - » Reduce seek time (more on that in a bit)
  - » Reduce time to allocate new blocks (look in smaller area)
  - » Spread many files over disk by spreading inodes (balance load)

When File Systems Go Bad

- File systems can have problems if the OS or disk fails
  - » Data in memory wasn't written out in time
  - » File operation was only partially completed
  - » Data on the disk was completely wiped out by disk failure
- Programs check for file system consistency
  - » Make sure every block is either free or in exactly one file
  - » Make sure directory structure is consistent
- Backup devices (tape, second disk, etc.) hold copies of data
  - » System utilities back up data on a regular basis
    - – Backup all files (occasionally)
    - – Backup modified files (more often)
  - » Data may be restored from backup if all else fails
  - » Files restored from backup if they're accidentally erased