Announcements

- Reading Chapter 11 (11.1-11.5)
  - suggested problems: 11.1, 11.2, 11.6, 11.8
- Midterm #2
  - it's next week (4/11/96)
  - don't forget to study synchronization

Filesystems

- Raw Disks can be viewed as:
  - a linear array of fixed sized units of allocation, called blocks
    - assume that blocks are error free (for now)
    - typical block size is 512 to 4096 bytes
  - can update a block in place, but must write the entire block
  - can access any block in any desired order
    - blocks must be read as a unit
    - for performance reasons may care about “near” vs. “far” blocks (but that is covered in a future lecture)
- A Filesystem:
  - provides a hierarchical namespace via directories
  - permits files of variable size to be stored
  - provides disk protection by restricting access to files based on permissions

File System Implementation

Application Programs

↓

Logical file system:
Knows about directories, application view of file names

↓

File Organization Module:
Can translate logical block addresses to physical block addresses

↓

Basic File System:
Issues physical block read/write commands

↓

Low Level I/O Control
Interfaces to hardware

Allocation Methods

- How do we select a free disk block to use?
- Contiguous allocation
  - allocate a contiguous chunk of space to a file
  - directory entry indicates the starting block and the length of the file
  - easy to implement, but
    - how to satisfy a given sized request from a list of free holes?
    - two options
      - first fit (find the first gap that fits)
      - best fit (find the smallest gaps that is large enough)
    - What happens if one wants to append to file?
  - from time to time, one will need to repack files

Linked Allocation

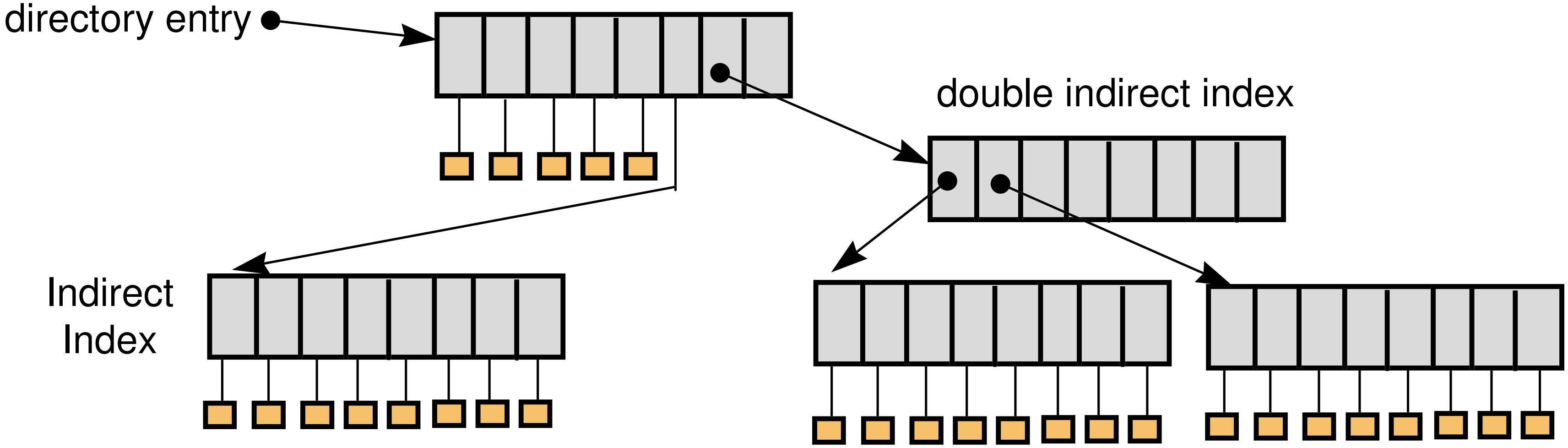
- Each file is a linked list of disk blocks, blocks can be located anywhere
  - Directory contains a pointer to the first and last block of a file
  - Each block contains a pointer to the next block
  - This is essentially a linked-list data structure
- Problems:
  - Best for sequential access data structures
    - requires sequential access whether you want to or not!
  - Reliability - one bad sector and all portions of your file downstream are lost
- Useful fix:
  - Maintain a separate data structure just to keep track of linked lists
  - Data-structure includes pointers to actual blocks

Indexed Allocation

- Bring all pointers together in an *index block*
  - Each file has its own index block - *i*th entry of index block points to *i*th block making up the file
- How large to make an index block?
  - unless one only wants to support fixed size files, index block scheme needs to be extensible
- Linked scheme:
  - maintain a linked list of indexed blocks
- Multilevel index:
  - Index block can point to other index blocks (which point to index blocks ....), which point to files
- Hybrid multi-level index
  - first n blocks are from a fixed index
  - next m blocks from an indirect index
  - next o blocks from a double indirect index

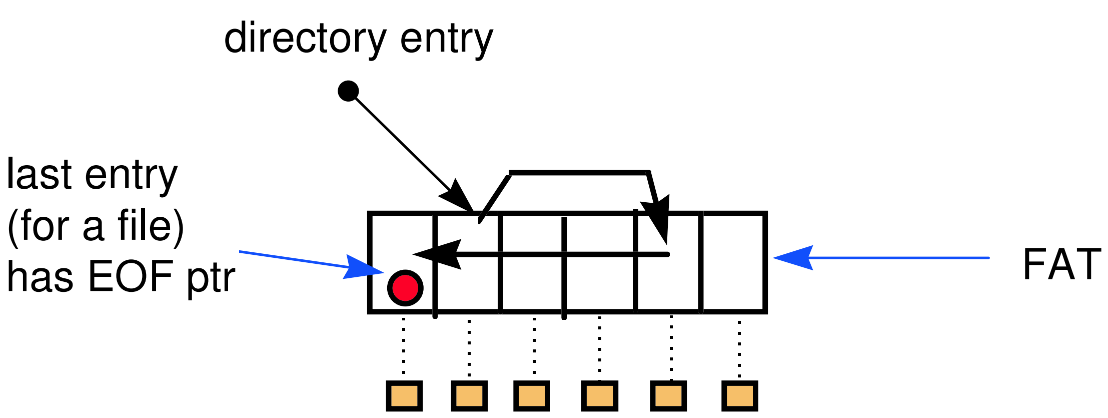
Hybrid Multi-level Index (UNIX)

- Observations
  - most files are small
  - most of the space on the disk is consumed by large files
- Want a flexible way to support different sized
  - assume 4096 byte block
  - first 12 blocks (48KB) are from a fixed index
  - next 1024 blocks (1MB) from an indirect index
  - next 1024^2 blocks (1GB) from a double indirect index
  - final 1024^3 blocks (1TB) from a triple indirect index



Modified Linked Allocation (FAT)

- Section of disk contains a table
  - called the file allocate table (FAT)
  - used in MS-DOS
- Directory entry contains the block number of the first block in the file
- Table entry contains the number of the next block in the file
- Last block has a end-of-file value as a table entry



i^{th} block corresponds to the i^{th} FAT entry

Performance Issues

- FAT
  - ✔ simple, easy to implement
  - ✔ faster to traverse than linked allocation
  - – random access requires following links
- Hybrid indirect
  - ✔ fast access to any part of the file
  - – more complex

Free Space Management

- How do we find a disk block to allocate?
- Bit Vectors
  - array of bits (one per block) that indicates if a block is free
  - compact so can keep in memory
    - 1.3 GB disk, 4K blocks -> 78K per disk
  - easy to find long runs of free blocks
- Linked lists
  - each disk block contains the pointer to the next free block
  - pointer to first free block is keep in a special location on disk
- Run length encoding (called counting in book)
  - pointer to first free block is keep in a special location on disk
  - each free block also includes a count of the number of consecutive blocks that are free

Implementing Directories

- Linear List
  - array of names for files
  - must search entire list to find or allocate a filename
  - sorting can improve search performance, but adds complexity
- Hash table
  - use hash function to find filenames in directory
  - needs a good hash function
  - need to resolve collisions
  - must keep table small and expand on demand since many directories are mostly empty

DOS Directories

- Root directory
  - immediately follows the FAT
- Directory is a table of 32 byte entries
  - 8 byte file name, 3 byte filename extension
  - size of file, data and time stamp, starting cluster number of the file, file attribute codes
  - Fixed size and capacity
- Subdirectory
  - This is just a file
  - Record of where the subdirectory is located is stored in the FAT

Unix Directories

- Space for directories are allocated in units called *chunks*
  – Size of a chunk is chosen so that each allocation can be transferred to disk in a single operation
  – Chunks are broken into variable-length directory entries to allow filenames of arbitrary length
  – No directory entry can span more than one chunk
  – Directory entry contains
    - pointer to inode (file data-structure)
    - size of entry
    - length of filename contained in entry (up to 255)
    - remainder of entry is variable length - contains file name

inodes

- File index node
- Contains:
  - Pointers to blocks in a file (direct, single indirect, double indirect, triple indirect)
  - Type and access mode
  - File’s owner
  - Number of references to file
  - Size of file
  - Number of physical blocks

Unix directories - links

- Each file has unique inode but it may have multiple directory entries in the same filesystem to reference inode
- Each directory entry creates a hard link of a filename to the file's inode
  - Number of links to file are kept in reference count variable in inode
  - If links are removed, file is deleted when number of links becomes zero
- Symbolic or soft link
  - Implemented as a file that contains a pathname
  - Symbolic links do not have an effect on inode reference count

Using UNIX filesystem data structures

- Example: find /usr/bin/vi
  - from Leffler, McKusick, Karels and Quarterman
  - Search root directory of filesystem to find /usr
    - root directory inode is, by convention, stored in inode #2
    - inode shows *where data blocks are* for root directory - *these blocks* (not the inode itself) *must* be retrieved and searched for entry user
    - we discover that the directory user’s inode is inode #4
  - Search user for bin
    - access blocks pointed to by inode #4 and search contents of blocks for entry that gives us bin’s inode
    - we discover that bin’s inode is inode #7
  - Search bin for vi
    - access blocks pointed to by inode #7 and search contents of block for an entry that gives us vi’s inode
    - we discover that vi’s inode is inode #7
  - Access inode #7 - this is vi’s inode