CIS520 – Operating Systems
Handout 13 – File System Implementation

- Discuss several file system implementation strategies.
- First implementation strategy: contiguous allocation. Just lay out the file in contiguous disk blocks. Used in VM/CMS - an old IBM interactive system. Advantages:
  - Quick and easy calculation of block holding data - just offset from start of file!
  - For sequential access, almost no seeks required.
  - Even direct access is fast - just seek and read. Only one disk access.

  Disadvantages:
  - Where is best place to put a new file?
  - Problems when file gets bigger – may have to move whole file!!
  - External Fragmentation.
  - Compaction may be required, and it can be very expensive.
- Next strategy: linked allocation. All files stored in fixed size blocks. Link together adjacent blocks like a linked list. Advantages:
  - No more variable-sized file allocation problems. Everything takes place in fixed-size chunks, which makes memory allocation a lot easier.
  - No more external fragmentation.
  - No need to compact or relocate files.

  Disadvantages:
  - Potentially terrible performance for direct access files - have to follow pointers from one disk block to the next!
  - Even sequential access is less efficient than for contiguous files because may generate long seeks between blocks.
  - Reliability -if lose one pointer, have big problems.
- FAT allocation. Instead of storing next file pointer in each block, have a table of next pointers indexed by disk block. Still have to linearly traverse next pointers, but at least don't have to go to disk for each of them. Can just cache the FAT table and do traverse all in memory. MS/DOS and OS/2 use this scheme.
- Table pointer of last block in file has EOF pointer value. Free blocks have table pointer of 0. Allocation of free blocks with FAT scheme is straightforward. Just search for first block with 0 table pointer.
- Indexed Schemes. Give each file an index table. Each entry of the index points to the disk blocks containing the actual file data. Supports fast direct file access, and not bad for sequential access.
- Question: how to allocate index table? Must be stored on disk like everything else in the file system. Have basically same alternatives as for file itself! Contiguous, linked, and multilevel index. In practice some combination scheme is usually used. This whole discussion is reminiscent of paging discussions.
- Will now discuss how traditional Unix lays out file system.

- First 8KB - label + boot block. Next 8KB - Superblock plus free inode and disk block cache.
- Next 64KB - inodes. Each inode corresponds to one file.
- Until end of file system - disk blocks. Each disk block consists of a number of consecutive sectors.
- What is in an inode - information about a file. Each inode corresponds to one file. Important fields:
  - Mode. This includes protection information and the file type. File type can be normal file (-), directory (d), symbolic link (l).
  - Owner
  - Number of links - number of directory entries that point to this inode.
  - Length - how many bytes long the file is.
  - Nblocks - number of disk blocks the file occupies.
  - Array of 10 direct block pointers. These are first 10 blocks of file.
  - One indirect block pointer. Points to a block full of pointers to disk blocks.
  - One doubly indirect block pointer. Points to a block full of pointers to blocks full of pointers to disk blocks.
  - One triply indirect block pointer. (Not currently used).

  So, a file consists of an inode and the disk blocks that it points to.
- Nblocks and Length do not contain redundant information - can have holes in files. A hole shows up as block pointers that point to block 0 - i.e., nothing in that block.
- Assume block size is 512 bytes (i.e. one sector). To access any of first 512*10 bytes of file, can just go straight from inode. To access data farther in, must go indirect through at least one level of indirection.
- What does a directory look like? It is a file consisting of a list of (name,inode number) pairs. In early Unix Systems the name was a maximum of 14 characters long, and the inode number was 2 bytes. Later versions of Unix removed this restriction, and each directory entry was variable length and also included the length of the file name.
- Why don't inodes contain names? Because would like a file to be able to have multiple names.
- How does Unix implement the directories . and ..? They are just names in the directory. . points to the inode of the directory, while .. points to the inode of the directory's parent directory. So, there are some circularities in the file system structure.
- User can refer to files in one of two ways: relative to current directory, or relative to the root directory. Where does lookup for root start? By convention, inode number 2 is the inode for the top directory. If a name starts with /, lookup starts at the file for inode number 2.
- How does system convert a name to an inode? There is a routine called namei that does it.
- Do a simple file system example, draw out inodes and disk blocks, etc. Include counts, length, etc.
- What about symbolic links? A symbolic link is a file containing a file name. Whenever a Unix operation has the name of the symbolic link as a component of a file name, it macro substitutes the name in the file in for the component.
- What disk accesses take place when list a directory, cd to a directory, cat a file? Is there any difference between ls and ls -F?
- What about when use the Unix rm command? Does it always delete the file? NO - it decrements the reference count. If the count is 0, then it frees up the space. Does this algorithm work for directories? NO - directory has a reference to itself (.). Use a different command.
- When write a file, may need to allocate more inodes and disk blocks. The superblock keeps track of data that help this process along. A superblock contains:

  - the size of the file system
  - number of free blocks in the file system
  - list of free blocks available in the file system
  - index of next free block in free block list
  - the size of the inode list
  - the number of free inodes in the file system
  - a cache of free inodes
  - the index of the next free inode in inode cache

  The kernel maintains the superblock in memory, and periodically writes it back to disk. The superblock also contains crucial information, so it is replicated on disk in case part of disk fails.

- When OS wants to allocate an inode, it first looks in the inode cache. The inode cache is a stack of free inodes, the index points to the top of the stack. When the OS allocates an inode, it just decrements index. If the inode cache is empty, it linearly searches inode list on disk to find free inodes. An inode is free iff its type field is 0. So, when go to search inode list for free inodes, keep looking until wrap or fill inode cache in superblock. Keep track of where stopped looking - will start looking there next time.

- To free an inode, put it in superblock's inode cache if there is room. If not, don't do anything much. Only check against the number where OS stopped looking for inodes the last time it filled the cache. Make this number the minimum of the freed inode number and the number already there.

- OS stores list of free disk blocks as follows. The list consists of a sequence of disk blocks. Each disk block in this sequence stores a sequence of free disk block numbers. The first number in each disk block is the number of the next disk block in this sequence. The rest of the numbers are the numbers of free disk blocks. (Do a picture) The superblock has the first disk block in this sequence.

- To allocate a disk block, check the superblock's block of free disk blocks. If there are at least two numbers, grab the one at the top and decrement the index of next free block. If there is only one number left, it contains the index of the next block in the disk block sequence. Copy this disk block into the superblock's free disk block list, and use it as the free disk block.

- To free a disk block do the reverse. If there is room in the superblock's disk block, push it on there. If not, write superblock's disk block into free block, then put index of newly free disk block in as first number in superblock's disk block.

- Note that OS maintains a list of free disk blocks, but only a cache of free inodes. Why is this?

  - Kernel can determine whether inode is free or not just by looking at it. But, cannot with disk block - any bit pattern is OK for disk blocks.
  - Easy to store lots of free disk block numbers in one disk block. But, inodes aren't large enough to store lots of inode numbers.
  - Users consume disk blocks faster than inodes. So, pauses to search for inodes aren't as bad as searching for disk blocks would be.
  - Inodes are small enough to read in lots in a single disk operation. So, scanning lists of inodes is not so bad.

- Synchronizing multiple file accesses. What should correct semantics be for concurrent reads and writes to the same file? Reads and writes should be atomic:

  - If a read execute concurrently, read should either observe the entire write or none of the write.
  - Reads can execute concurrently with no atomicity constraints.

- How to implement these atomicity constraints? Implement reader-writer locks for each open file. Here are some operations:

  - Acquire read lock: blocks until no other process has a write lock, then increments read lock count and returns.
  - Release read lock: decrements read lock count.
  - Acquire write lock: blocks until no other process has a write or read lock, then sets the write lock flag and returns.
  - Release write lock: clears write lock flag.

- Obtain read or write locks inside the kernel's system call handler. On a Read system call, obtain read lock, perform all file operations required to read in the appropriate part of file, then release read lock and return. On Write system call, do something similar except get write locks.

- What about Create, Open, Close and Delete calls? If multiple processes have file open, and a process calls Delete on that file, all processes must close the file before it is actually deleted. Yet another form of synchronization is required.

- How to organize synchronization? Have a global file table in addition to local file tables. What does each file table do?

  - Global File Table: Indexed by some global file id - for example, the inode index would work. Each entry has a reader/writer lock, a count of number of processes that have file open and a bit that says whether or not to delete the file when last process that has file open closes it. May have other data depending on what other functionality file system supports.
  - Local File Table: Indexed by open file id for that process. Has a pointer to the current position in the open file to start reading from or writing to for Write and Read operations.

- For your nachos assignments, do not have to implement reader/writer locks - can just use a simple mutual exclusion lock.

- What are sources of inefficiency in this file system? Are two kinds - wasted time and wasted space.

- Wasted time comes from waiting to access the disk. Basic problem with system described above: it scatters related items all around the disk.

  - Inodes separated from files.
  - Inodes in same directory may be scattered around in inode space.
  - Disk blocks that store one file are scattered around the disk.

  So, system may spend all of its time moving the disk heads and waiting for the disk to revolve.

- The initial layout attempts to minimize these phenonmena by setting up free lists so that they allocate consecutive disk blocks for new files. So, files tend to be consecutive on disk. But, as use file system, layout gets scrambled. So, the free list order becomes increasingly randomized, and the disk blocks for files get spread all over the disk.

- Just how bad is it? Well, in traditional Unix, the disk block size equaled the sector size, which was 512 bytes. When they went from 3BSD to 4.0BSD they doubled the disk block size. This more than doubled the disk performance. Two factors:

  - Each block access fetched twice as much data, so amortized the disk seek overhead over more data.
  - The file blocks were bigger, so more files fit into the direct section of the inode index.

  But, still pretty bad. When file system first created, got transfer rates of up to 175 KByte per second. After a few weeks, deteriorated down to 30 KByte per second. What is worse, this is only about 4 percent (!!!!) of maxmimum disk throughput. So, the obvious fix is to make the block size even bigger.

- Wasted space comes from internal fragmentation. Each file with anything in it (even small ones) takes up at least one disk block. So, if file size is not an even multiple of disk block size, there will be wasted space off the end of the last disk block in the file. And, since most files are small, there may not be lots of full disk blocks in the middle of files.

- Just how bad is it? It gets worse for larger block sizes. (so, maybe making block size bigger to get more of the disk transfer rate isn't such a good idea...). Did some measurements on a file system at Berkeley, to calculate size and percentage of waste based on disk block size. Here are some numbers:

Space Used (Mbytes)	Percent Waste	Organization
775.2	0.0	Data only, no separation between files
828.7	6.9	Data + inodes, 512 byte block
866.5	11.8	Data + inodes, 1024 byte block
948.5	22.4	Data + inodes, 2048 byte block
1128.3	45.6	Data + inodes, 4096 byte block

- Notice that a problem is that the presence of small files kills large file performance. If only had large files, would make the block size large and amortize the seek overhead down to some very small number. But, small files take up a full disk block and large disk blocks waste space.
- In 4.2BSD they attempted to fix some of these problems.
- Introduced concept of a cylinder group. A cylinder group is a set of adjacent cylinders. A file system consists of a set of cylinder groups.
- Each cylinder group has a redundant copy of the super block, space for inodes and a bit map describing available blocks in the cylinder group. Default policy: allocate 1 inode per 2048 bytes of space in cylinder group.
- Basic idea behind cylinder groups: will put related information together in the same cylinder group and unrelated information apart in different cylinder groups. Use a bunch of heuristics.
- Try to put all inodes for a given directory in the same cylinder group.
- Also try to put blocks for one file adjacent in the cylinder group. The bitmap as a storage device makes it easier to find adjacent groups of blocks. For long files redirect blocks to a new cylinder group every megabyte. This spreads stuff out over the disk at a large enough granularity to amortize the seek time.
- Important point to making this scheme work well - keep a free space reserve (5 to 10 percent). Once above this reserve, only supervisor can allocate disk blocks. If disk is almost completely full, allocation scheme cannot keep related data together and allocation scheme degenerates to random.
- Increased block size. The minimum block size is now 4096 bytes. Helps read bandwidth and write bandwidth for big files. But, don't waste a lot of space for small files? Solution: introduce concept of a disk block fragment.
- Each disk block can be chopped up into 2, 4, or 8 fragments. Each file contains at most one fragment which holds the last part of data in the file. So, if have 8 small files they together only occupy one disk block. Can also allocate larger fragments if the end of the file is larger than one eighth of the disk block. The bit map is laid out at the granularity of fragments.
- When increase the size of the file, may need to copy out the last fragment if the size gets too big. So, may copy a file multiple times as it grows. The Unix utilities try to avoid this problem by growing files a disk block at a time.
- Bottom line: this helped a lot - read bandwidth up to 43 percent of peak disk transfer rate for large files.
- Another standard mechanism that can really help disk performance - a disk block cache. OS maintains a cache of disk blocks in main memory. When a request comes, it can satisfy request locally if data is in cache. This is part of almost any IO system in a modern machine, and can really help performance.
- How does caching algorithm work? Devote part of main memory to cached data. When read a file, put into disk block cache. Before reading a file, check to see if appropriate disk blocks are in the cache.
- What about replacement policy? Have many of same options as for paging algorithms. Can use LRU, FIFO with second chance, etc.

- How easy is it to implement LRU for disk blocks? Pretty easy - OS gets control every time disk block is accessed. So can implement an exact LRU algorithm easily.
- How easy was it to implement an exact LRU algorithm for virtual memory pages? How easy was it to implement an approximate LRU algorithm for virtual memory pages?
- Bottom line: different context makes different cache replacement policies appropriate for disk block caches.
- What is bad case for all LRU algorithms? Sequential accesses. What is common case for file access? Sequential accesses. How to fix this? Use free-behind for large sequentially accessed files - as soon as finish reading one disk block and move to the next, eject first disk block from the cache.
- So what cache replacement policy do you use? Best choice depends on how file is accessed. So, policy choice is difficult because may not know.
- Can use read-ahead to improve file system performance. Most files accessed sequentially, so can optimistically prefetch disk blocks ahead of the one that is being read.
- Prefetching is a general technique used to increase the performance of fetching data from long-latency devices. Can try to hide latency by running something else concurrently with fetch.
- With disk block caching, physical memory serves as a cache for the files stored on disk. With virtual memory, physical memory serves as a cache for processes stored on disk. So, have one physical resource shared by two parts of system.
- How much of each resource should file cache and virtual memory get?
  - Fixed allocation. Each gets a fixed amount. Problem - not flexible enough for all situations.
  - Adaptive - if run an application that uses lots of files, give more space to file cache. If run applications that need more memory, give more to virtual memory subsystem. Sun OS does this.
- How to handle writes. Can you avoid going to disk on writes? Possible answers:
  - No - user wants data on stable storage, that's why he wrote it to a file.
  - Yes - keep in memory for a short time, and can get big performance improvements. Maybe file is deleted, so don't ever need to use disk at all. Especially useful for /tmp files. Or, can batch up lots of small writes into a larger write, or can give disk scheduler more flexibility.

  In general, depends on needs of the system.
- One more question - do you keep data written back to disk in the file cache? Probably - may be read in the near future, so should keep it resident locally.
- One common problem with file caches - if use file system as backing store, can run into double caching. Eject a page, and it gets written back to file. But, disk blocks from recently written files may be cached in memory in the file cache. In effect, file caching interferes with performance of the virtual memory system. Fix this by not caching backing store files.
- An important issue for file systems is crash recovery. Must maintain enough information on disk to recover from crashes. So, modifications must be carefully sequenced to leave disk in a recoverable state at all times.