**File Organization Part 1 – Storage, Record Formats, and Linux**

|  |  |
| --- | --- |
| **File Organization**  Since data is important to Database Management, we begin our journey with data files. We will discuss:   * Types of Storage * Record Formats * Linux Files, inode, and file systems * Sequential File Organization * Direct File Organization * Indexed File Organization |  |
| **Types of Storage**  **Primary Storage**   * main memory and cache memory * fast access by the CPU * most expensive per byte of any of the storage * usually volatile   **Secondary Storage**   * Magnetic hard disk drives (HDD) and solid-state drive (SSD) * provides read/write access that persists; non-volatile * data retrieved in blocks * Cannot be accessed directly by CPU * Active databases are placed on secondary storage   **Secondary Storage - Magnetic Hard Disks**   * Stack of rotating record surfaces (i.e., platters) * Each platter contains many tracks. A track is a ring. Tracks don't spiral. * Disk arm moves to access a cylinder which consists of one track from each platter * **Seek time**  - time to move disk arm to desired cylinder * **Rotational latency –** time to wait for desired block to move under read/write head * **Transfer Time –** time to move data to/from disk   **Hard Disk Drives** used by businesses doing *many* transactions have   * low seek times – less than 3 ms * low rotational latency – less than 2ms * high transfer rates – more than 200MB/s * redundancy – reduce impact of hardware failure * Average request for one block can take less than 10ms   There are now 1TB hard drives for as low as $50. Sure, the price seems great so why don't businesses just use these?   * Motor start up latency may be as bad as 3-5 secs * Poor transfer rates 10-50 MB/s * Seek time – 10ms+ * much higher rate of failure | Diagram 1: Disk Drive with Arm. Part B shows dual-sided platters.  fig16_01.jpg  Diagram 2: Solid-State Drive (SSD)  Image result for emc solid state drive |
| **Types of Storage Continued**  **Tertiary Storage**   * optical disks (CD-ROMs, DVDs) and tapes * Larger capacities and cost less than secondary storage * Slower access speeds than secondary storage * Cannot be accessed directly by CPU * Typically, first copied to secondary storage before providing to CPU   Many devices are huge optical juke boxes or tape libraries capable of managing 1000s of media accessed by robot arms that place the media in a selected drive. | Image result for tertiary storage devices |
| **Record Formats** FIXED UNBLOCKED – Logical Record = Physical Record   record identifier  condition flag | |
| FIXED BLOCKED – Multiple Logical Records in a Physical Record | |
| VARIABLE UNBLOCKED – One variable-sized logical per physical   Note that the length values include the size of the length fields. | |
| VARIABLE BLOCKED – Multiple variable-sized logical per physical | |
| Advantage of blocking   * fewer control areas * for variable length, fewer block lengths * can save time when I/O is sequential or clustered |  |
| **Linux Files**  We have thought of text files as simply contiguous bytes with text lines separated by line feeds in Unix. That is a simple high-level perspective. We will now look under the covers of files in Unix.  Unix manages file storage as **blocks** (typically 4096 bytes). One file can consist of many blocks which are usually not contiguous. For a text file, many text lines can be in a block. Some text lines might actually span across one of more blocks. Why is that necessary? Is it only because of a text line that is bigger than the block size?  Linux itself considers the blocks to be fixed unblocked; however, applications can choose to place multiple logical records in a block. | The blue blocks are for one file in this diagram. |
| **Linux Inode**  Unix needs a mechanism to keep track of the blocks that make up a file. Why don't the blocks simply have a pointer to the next block? That would work for simple stream files.  Suppose our file has many customer records. If we know that customer 123123's record is at byte offset 10,350,800, would it make since to have to read through over 2500 blocks to get to that customer's data assuming 4K per block?  Unix (and of course Linux) use an **inode** for each file. It contains an initial block which references other blocks (using their addresses on the storage) in a manner like an array of pointers.  An inode contains   * stat info   + size of the file in bytes   + device ID of the device containing the file   + user ID of the owner   + group ID   + file mode containing the file type and the permissions mode (3 octal value)   + timestamps for when the inode was last changed, content last modified, and last access   + link count * index of pointers:   + 12 **direct pointers** to data blocks   + **one indirect pointer** that points to a block of data pointers   + **one double indirect pointer** that points to a block of pointers which then points to blocks of data pointers   + **one triple indirect pointer** that points to a block of pointers that points to blocks of pointers which point to blocks of pointers which point to data blocks * Note that an inode does not include the name of the file.   The **indirect nodes** contain either pointers to data blocks or pointers to indirect nodes. The number of entries is dependent on the block size for the file system and the size of a block address (either 4 or 8 bytes). | Assuming **4 byte** block addresses and **4096 byte** blocks, how big of a file can we possibly reference with this inode approach? Answer the following:  1. How much data can be referenced by the total from the 12 direct pointers?   * Number of entries is 12 * Size in bytes is 12 \* 4096 = 49,152 data bytes   2. How much data can be referenced by the 12 direct pointers and the additional one indirect pointer?   * The indirect node would contain 4096 bytes/block \* 1 entry /4 bytes = 1024 entries/block. * Total entries so far is ?? * Size in bytes is ?? \* 4096 = 4,243,456 data bytes (4mb)   3. How much data can be referenced by the 12 direct pointers, the one indirect pointer, and the one double indirect pointer?   * The first double indirect node has 1024 entries. It can reference 1024 indirect blocks. Therefore, 1024\*1024 double indirect index entries = 1,048,576 entries. * Total entries is 1,048,576 entries + 1036 entries (indirect, direct) * 1,049,612 entries \* 4096 = 4,299,210,752 bytes (4 gb)   4. How much data can be referenced as a total?   * Number of entries in the last level of indexes =  10243= 1,073,741,824 * Total entries is 1,073,741,824 + 1,049,612 * 1,074,791,436 \* 4096 = 4,402,345,721,856 bytes = 4TB. |
|  |  |
| **Exercise**  To access a single particular data block (just one) if our file is 1TB, what is the maximum number of reads necessary? | Based on the size, do we need the inode's indirect? double indirect? triple indirect? ??  What is the worst case for reading one data block in our file?  ??  Reads:   * ?? - for iNode (which is frequently in memory if heavily used)   ?? - index node reads  ?? - data read  Worst case is ?? reads. |
| **Linux File System**  Physical disks are organized into **file systems** to provide logical access to the actual bytes of a file. The file system controls how data is stored and retrieved.  To logically access data in a file, we specify a filename and a relative byte offset. Using the file system, Linux translates those into the actual physical location.  The file system is formatted to contain:  **superblock** size of file system, number of inodes, number of data blocks, block size  **allocation bitmaps** free and used inodes, free and used data blocks  **inode blocks** array of inode blocks  **data blocks** used for data blocks | Formatted file system:    The superblock and the bitmaps are usually kept in memory during execution. |
|  |  |

**©2017 Larry W. Clark,** UTSA CS students may make copies for their personal use