**I/O Part 1 Overview**

|  |  |
| --- | --- |
| System Call - request to the kernel layer to perform a task that must be accomplished by the operating system.  Some categories of system calls:  I/O   * creating files and directories * reading/writing files * positioning in files * reading directories   Process Control   * creating processes and threads * terminating processes and threads * obtaining process ID   Communication   * manipulating pipes * manipulating sockets * managing shared memory | Linux layers diagram    This course will emphasize **shells**, **utilities**, and **system calls**. |
| In CS1713 and CS 2123, you used **standard stream I/O functions** (scanf, fgets, printf, fopen, fclose). These functions were intended to make I/O easier than the low level functions by handling:   * data conversions (e.g., printf format codes convert from C data types to character output) * buffer input data, allowing multiple calls to access data that is physically read with a single input request * buffer output data, allowing multiple calls to print data although a physical write is done for larger quantities * efficiently read/write data   Under the covers of those functions are **low level I/O operations** which do the real I/O work.  We will discuss the internal representations of files and directories, standard I/O functions, file descriptors, and then discuss the low level I/O operations. | |  |  |  | | --- | --- | --- | | **Function** | **Category** | **Purpose** | | **scanf**  **fscanf**  **sscanf** | std i/o (stream) | stream input using format codes.  Note that sscanf gets its data from a string variable. | | **gets**  **fgets** | std i/o (stream) | stream input of text lines | | **printf**  **fprintf** | std i/o (stream) | stream output using format codes | | **getc**  **fgetc** | std i/o (stream) | get next char from a stream | | **putc**  **fputc** | std i/o (stream) | put a char to a stream | | **fopen** | std i/o | open a file for buffered i/o | | **fclose** | std i/o | close a file opened by fopen | | **fread** | binary read | binary input of one or more logical records | | **fwrite** | binary write | binary output of one or more logical records | | **fseek** | binary position | changes file position to a location relative to a number of bytes from the beginning of the file | | **open** | unix low level i/o | opens a file | | **close** | unix low level i/o | closes a file | | **read** | unix low level i/o | reads a specified number of bytes at the current position | | **write** | unix low level i/o | writes a specified number of bytes at the current position | | **lseek** | unix low level i/o | similar to fseek | | **stat**  **fstat** | unix low level i/o | returns the stat structure for a file which includes inodeNr, file type, file mode, number of links, size, etc. | | **opendir** | unix low level i/o | opens the specified directory for reading | | **readdir** | unix low level i/o | reads the next directory entry. | | **closedir** | unix low level i/o | closes a directory | |
| **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? ?? | The blue blocks are for one file in this diagram. |
| **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). | All the green blue and white are just blocks and blocks are 4k in size  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 = 48k (49152)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 is 1024 + 12 * Size in bytes is 1036\* 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. <- This is the maximum size of the file mostly |
| **Exercise**  To access a single particular data block (just one) if our file is 1TB, what is the maximum number of reads necessary? 5 because 1 TB mean we are accessing the triple indirect: so go back to the graph to see the path of triple indirect path | We would need the triple indirect pointers due to the size. Worst case is with the triple indirect.  Reads:   * 1 - for iNode (which is frequently in memory if heavily used) * 3 - index node reads since triple indirect * 1 - data read   Worst case is 5 reads. |
| **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. |
| **Directories**  There can be many directories on a file system. We have already discussed that Linux uses a hierarchical directory structure, but what is the actual implementation of a directory? A directory is a single file which contains a list of directory entries:  **file name** name of a file  **inode** index into an array of inodes  Effectively, a directory maps file names to inode values. Once we have a file's inode, we can then reference any of its data. To allow the internal implementation of a directory file to be implementation independent, Linux provides functions for creating, reading, and removing directories.    Where does Linux know the type of file (e.g., directory, simple file, link, pipe, socket)? Is it in the directory? Stat info |  |
| **Links**  Up until now, files are only referenced within their parent directory. Files can be referenced in multiple places through the use of Links.  A **symbolic link**, or symlink, is a file with its own inode which points directly to another file (something like a shortcut in Windows) while a **hard link** shares the file’s inode. If the original file to which a link is pointing is deleted, the symlink will remain but no longer work since it points to nothing. The hardlink will still point to the file even though the original inode is no longer referencing it.  You can delete a softlink without deleting the actual file since the inode of the file is different than the inode of the link. The softlink is simply a file which contains a path to the target file.  Creating a hard link is like creating a new file which points to the same data as another file. | **Creating a symlink**  ln -s <file> <linkname>  **Creating a hard link**  ln <file> <linkname>  symlink is like a shortcut on window. If the original file was to be deled the symlink is a broken link  hard link is like 2 pointers point to the same obj(inode) when we get rif of 1 ptr the other one still point to the obj |

**©2017-18 Larry W. Clark and Rocky Slavin,** UTSA CS students may make copies for their personal use