Notes on files and system calls. CSE 333 3/13/19 Perkins

Goal: get a better understanding about how the OS and C libraries interact for file I/O and system calls plus where/when buffering happens, what's a file, etc.

Disclaimer: this is conceptually on target, but detailed implementation is likely different and/or more complex.

## What's on disk

What's a file? First approximation is a sequence of 4K blocks of bytes + metadata. Individual file is represented by an inode that points to the blocks and contains owner info, etc.

Directories: special files in the file system that point to other directories and to inodes of files in those directories. This graph forms a DAG starting at / (i.e., cycles make no sense and are not allowed).

## POSIX (OS level) I/O

Walk through what happens when a user process does I/O directly through the OS open / read / write / close interface. No user-level buffering.

Linux Kernel data structures:

Process table. Global table with one entry for each process in the system.

- Each process table entry contains (or points to) a process-unique table of file descriptors used by that process. For each open file there is a pointer from the fd entry to the system-wide file table entry for that file

File table. Global table with one entry for each file currently open by some process. If more than one process has the same file open, there is a separate entry in this table for each process unless the process cloned or duped the file descriptor, in which case those descriptors point to the same file table entry.

- File table entry contains information about the current offset in the file for this process and a pointer to the actual file data with inode information, etc. read from disk (oversimplified, but roughly correct).

Buffer cache. A collection of disk-block size buffers plus identifying data. All data read/written to files flows through the buffer cache.

Operations:

*fd = open(filepath, flags)*

- Convert *filepath* to an inode number (actual file on disk) by walking directory tree as needed. (Much of this data is probably cached, but potentially requires one disk access for each directory level.)
- Allocate a (system) file table entry and store inode information, file offset, mode, etc.
- Allocate next available entry in process file descriptor (fd) table and initialize with pointer to file table entry
- If open for writing and truncating, discard existing blocks in the file
- Return index in process fd table as result of open system call (this *fd* is used in later I/O function calls)

*read(fd, buffer, size)* [ignoring eof detection for simplicity]

- Use *fd* to retrieve file table entry for file
- Calculate block number and offset in file
- Find block in buffer cache. If block is not present in the buffer cache, read from disk and suspend process until data available
- Copy data from buffer cache block to user buffer and update current file read position in file table entry. Repeat as needed if request spans multiple disk blocks.

Notes:
- If file system detects sequential reading, it may (probably will) read subsequent blocks into the buffer cache ahead of time so the data is more likely to be available immediately when needed by a later read operation
- I/O requests that start on file block boundaries and have lengths that are multiples of the block size may be significantly more efficient than writes for partial blocks or writes that span block boundaries

*write(fd, buffer, size)* [similar to read]

- Use *fd* to retrieve file table entry for file
- Calculate block number and offset
- Locate block in buffer cache
  - If block is already allocated as part of the file but not present in the buffer cache, read existing block from disk
  - If writing new data at end of file, allocate new empty block in buffer cache if new data starts a new block, otherwise read existing block where data should be appended
- Write data into buffer cache

Buffer cache blocks containing new data will eventually be written to disk. When?

- Retain block in cache in case additional data is written to it soon
- Write inactive but modified blocks to disk if running out of available space in the buffer cache and blocks need to be allocated for other I/O operations
- *sync()* schedules all blocks containing new data to be written to disk. Eventually. A system-wide *sync()* is done periodically to schedule blocks with new/modified data for writing. No guarantees on when the actual write is done, but it will be relatively soon. Leave each block in the cache even after it is written if there's room. This speeds up future access to the data if the block is used again.
- *fsync(fd)* waits for all blocks with new data associated with *fd* to be written to disk.

*close(fd)*

- If this is the last *fd* referencing the system file table entry for this file, free the system file table entry
- Free the *fd* entry in the process file descriptor table

Note that this does not guarantee that dirty (modified) buffer cache blocks are written to disk immediately. It also does not immediately free blocks in the cache until the space is needed for something else. That will speed things up if the file is re-opened again soon by the same or some other process.

Phew! That's a lot.

But wait, there's more….

Standard I/O Library (glibc stdio)

Provides a portable, richer set of I/O functions – stream abstraction, formatted text I/O, including character <-> binary conversions, etc.

Implemented on top of POSIX I/O layer in the OS, which does the actual I/O.

Data structures in user address space:

FILE: a struct allocated on the user process heap. One of these per open file. Contains system file descriptor (*fd*) number, status information, pointer to stdio (user address space) library buffers for this stream, current position information in the buffers and file, etc.

Stdio-library managed buffers: data being read or written is accumulated here. Allocated on the heap and typically the same size as disk blocks (for efficiency).

Buffering:

Three basic options. These control how the user-space stdio buffers are handled. When a read or write is issued to the OS (POSIX) level, other buffering involving the system buffer pool, which is not directly visible to the user-level code, will happen, but at user level, there are these possibilities:

- Full buffering: data is accumulated in the stdio buffers and actual I/O is done when a block is full (writing) or new data is needed (reading)
- Line buffering: data is accumulated until a newline is encountered, then a read or write happens immediately. Used for terminal I/O & we'll ignore for now (look up details when you need them)
- Unbuffered: Each stdio I/O operation is passed directly to the OS level immediately for each read or write. Bypasses the extra user-space buffering overhead but at the expense of a system call on each I/O operation. Will slow things down compared to buffered I/O unless the program is minimizing the number of system calls it executes in some way.

Functions:

*FILE * fopen(path, mode)*

- Allocate a new FILE struct and stdio buffer for this stream on the heap; initialize FILE struct
- Execute *fd=open(path, …)* to actually open the file and store returned *fd* in FILE struct
- Return a pointer to the FILE struct

*Input operations (fgetc, fgets, fread, fscanf, etc.)*

- If necessary data is not available in the stdio buffer, use *read(fd)* to read file block(s) as needed from the OS into user address space. Not needed if a partially read block is still available in user space and additional data can be read from there.
- Copy/process appropriate amount of data from the user buffer and return results to user. May involve converting from file characters to binary values.

*Output operations (fputc, fputs, fwrite, fprintf, etc.)*

- Do necessary conversions to characters
- Copy data to stdio buffer. If buffer has no more room, use *write(fd)* to actually write the data to the OS (where it probably will be buffered, then eventually written to disk)

*fflush(FILE *f)*

- Force a write of all buffered data using *write(fd)*

*fclose(FILE *f)*

- Flush any buffered output data with *write(fd)* to OS/POSIX level
- Do a *close(fd)* on the underlying file
- Free all allocated user-space memory buffers and the FILE struct for this stream

Efficiency/Performance notes

- Access to data in main memory is orders of magnitude faster than if the data needs to be retrieved from disk.  That is the rationale behind the OS/POSIX buffer cache – keep active file pages in memory where the data can be accessed quickly.
- Ordinary function calls to either user or stdio library functions are significantly faster than system calls.  That's why the stdio library reads/writes entire disk blocks on each read/write call to the system and handles individual fread/fwrite opeations out of a local buffer in the user address space.  The individual fread/fwrite operations using the local buffer are faster than would be needed if each of those operations involved a system call to do a read/write directly using the OS buffer pool.

<u>References</u>

*The C Programming Language,* Kernighan & Ritchie, Prentice-Hall, 2nd ed, 1988 (ch. 8 especially)
*The Design of the Unix Operating System,* Maurice Bach, Prentice-Hall, 1986
*The Linux Programming Interface,* Michael Kerrisk, No Starch Press, 2010
*Advanced Programming in the Unix Enviornment*, Stevens & Rago, Addison-Wesley, 3rd ed, 2013