Lecture Notes on Operating Systems
Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

The xv6 filesystem

1. Device Driver

- The data structure `struct buf` (line 3750) is the basic unit of storing a block's content in the xv6 kernel. It consists of 512 bytes of data, device and sector numbers, pointers to the previous and next buffers, and a set of flags (valid, dirty, busy).
- Sheets 41 and 42 contain the device driver code for xv6. The disk driver in xv6 is initialized in `ideinit` (line 4151). This function enables interrupts on last CPU, and waits for disk to be ready. It also checks if a second disk (disk 1) is present.
- The list `idequeue` is the queue of requests pending for the disk. This list is protected by `idelock`.
- The function `iderw` (line 4254) is the entry point to the device driver. It gets requests from the buffer cache / file system. It's job is to sync a specified buffer with disk. If the buffer is dirty, it must be written to the disk, after which the dirty flag must be cleared and the valid flag must be set. If the buffer is not dirty, and not valid, it indicates a read request. So the specified block is read into the buffer from the disk, and valid flag is set. If the buffer is valid and not dirty, then there is nothing to do. This function does not always send the requests to the disk, because the disk may be busy. Instead, it adds request to queue. If this is the first request in the queue, then it starts the requested operation and blocks until completion.
- The function `idestart` (line 4175) starts a new disk request—the required information to start the operation (block number, read/write instructions) are all provided to the driver. The process calling `idestart` will eventually sleep waiting for the operation to finish.
- The function `ideintr` (line 4202) is called when the requested disk operation completes. It updates flags, wakes up the process waiting for this block, and starts the next disk request.

2. Buffer cache

- The code for the buffer cache implementation in xv6 can be found in sheets 43 and 44. The buffer cache is simply an array of buffers initialized at startup in the main function, and is used to cache disk contents.

- Any code that wishes to read a disk block calls `bread`. This function first calls the function `bget`. `bget` looks for the particular sector in the buffer cache. There are a few possibilities. (i) If the given block is in cache and isn't being used by anyone, then `bget` returns it. Note that this can be clean/valid or dirty/valid (i.e., someone else got it from disk, and maybe even wrote to it, but is not using it). (ii) If it is in cache but busy, `bget` sleeps on it. After waking up, it tries to lock it once again by checking if the buffer corresponding to that sector number exists in cache.(Why can't it assume that the buffer is still available after waking up? Because someone else could have reused that slot for another block, as described next.) (iii) If the sector number is not in cache, it must be fetched from disk. But for this, the buffer cache needs a free slot. So it starts looking from the end of the list (LRU), finds a clean, non busy buffer to recycle, allots it for this block, marks it as invalid (since the contents are of another block), and returns it. So when `bread` gets a buffer from cache that is not valid, it will proceed to read from disk.

- A process that locked the buffer in the cache may make changes to it. If it does so, it must flush the changes to disk using `bwrite` before releasing the lock on the buffer. `bwrite` first marks the buffer as dirty (so that the driver knows it is a write operation), and then calls `iderw` (which adds it to the request queue). Note that there is no guarantee that the buffer will be immediately written, since there may be other pending requests in the queue. Also note that the process still holds the lock on the buffer, as the busy flag is set, so no other process can access the buffer.

- When a process is done using a disk block, it calls `brelse`. This function releases the lock by clearing the busy flag, so other processes sleeping on this block can start using this disk block contents again (whether clean or dirty). This function also moves this block to the head of the buffer cache, so that the LRU replacement algorithm will not replace it immediately.

3. Logging

- Sheets 45–47 contain the code for transaction-based logging in xv6. A system call may change multiple data and metadata blocks. In xv6, all changes of one system call must be wrapped in a transaction and logged on disk, before modifying the actual disk blocks. On boot, xv6 recovers any committed but uninstalled transactions by replaying them. In the boot process, the function `initlog` (line 4556) initializes the log and does any pending recovery. Note that the recovery process ignores uncommitted transactions.

- A special space at the end of the disk is dedicated for the purpose of storing the log of uncommitted transactions, using the data structure `struct log` (line 4537). The `committing` field indicates that a transaction is committing, and that system calls should wait briefly. The field `outstanding` denotes the number of ongoing concurrent system calls—the changes from several concurrent system calls can be collected into the same transaction. The `struct logheader` indicates which buffers/sectors have been modified as part of this transaction.

- Any system call that needs to modify disk blocks first calls `begin_op` (line 4628). Thus function waits if there is not enough space to accommodate the log entry, or if another transaction is committing. After beginning the transaction, system call code can use `bread` to get access to a disk buffer. Once a disk buffer has been modified, the system call code must now call `log_write` instead of `bwrite`.This function updates the block in cache as dirty, but does not write to disk. Instead, it updates the transaction header to note this modified log. Note that this function absorbs multiple writes to the same block into the same log entry. Once the changes to the block have been noted in the transaction header, the block can be released using `brelse`. This process of `bread`, `log_write`, and `brelse` can be repeated for multiple disk blocks in a system call.

- Once all changes in a transaction are complete, the system call code must call `end_op` (line 4653). This function decrements the outstanding count, and starts the commit process once all concurrent outstanding system calls have completed. That is, the last concurrent system call will release the lock on the log, set the commit flag in the log header, and start the long commit process. The function `commit` (line 4701) first writes all modified blocks to the log on disk (line 4683), then writes the log header to disk at the start of the log (4604), then installs transactions from the log one block at a time. These functions that actually write to disk now call `bwrite` (instead of the system call code itself calling them). Once all blocks have been written to their original locations, the log header count is set to zero and written to disk, indicating that it has no more pending logged changes. Now, when recovering from the log, a non-zero count in the log header on disk indicates that the transaction has committed, but hasn't been installed, in which case the recovery module installs them again. If the log header count is zero, it could mean that only part of the transactions are in the log, or that the system exited cleanly, and nothing needs to be done in both cases.

4. Inodes

- Sheets 39 and 40 contain the data structures of the file system, the layout of the disk, the inode structures on disk and in memory, the directory entry structure, the superblock structure, and the `struct file` (which is the entry in the open file table). The inode contains 12 direct blocks and one single indirect block. Also notice the various macros to computes offsets into the inode and such. Sheets 47–55 contain the code that implements the core logic of the file system.

- In xv6, the disk inodes are stored in the disk blocks right after the super block; these disk inodes can be free or allocated. In addition, xv6 also has an in-memory cache of inodes, protected by the `icache.lock`. Kernel code can maintain C pointers to active inodes in memory, as part of working directories, open files etc. An inode has two counts associated with it: `nlink` says how many links point to this file in the directory tree. The count `ref` that is stored only in the memory version of the inode counts how many C pointers exist to the in-memory inode. A file can be deleted from disk only when both these counts reach zero.

- The function `readsb` (4777) reads superblock from disk, and is used to know the current state of disk. The function `balloc` (line 4804) reads the free block bit map, finds a free block, fetches is into the buffer cache using `bread`, zeroes it, and returns it. The function `bfree` (line 4831) frees the block, which mainly involves updating the bitmap on disk. Note that all of these functions rely on the buffer cache and log layers, and do not access the device driver or disk directly.

- The function `ialloc` (line 4953) allocates an on-disk inode for a file for the first time. It looks over all disk inodes, finds a free entry, and gets its number for a new file. The function `iget` (line 5004) tries to find an in-memory copy of an inode of a given number. If one exists (because some one else opened the file), it increments the reference count, and returns it. If none exists, it finds an unused empty slot in the inode cache and returns that inode from cache. Note that iget is not guaranteeing that the in-memory inode is in sync with the disk version. That is the job of the function `ilock` (line 5053), which takes an in memory inode and updates its contents from the corresponding disk inode. So, to allocate a new inode to a file, one needs to to allocate an on-disk inode, an in-memory inode in the cache, and sync them up.

- Why do we need two separate functions for the logic of `iget` and `ilock`? When a process locks an inode, it can potentially make changes to the inode. Therefore, we want to avoid several processes concurrently accessing an inode. Therefore, when one process has locked an inode in the cache, another process attempting to lock will block. However, several processes can maintain long term pointers to an inode using iget, without modifying the inode. As long as a C pointer is maintained, the inode won't be deleted from memory or from disk: the pointers serve to keep the inode alive, even if they do not grant permission to modify it.

- When a process locks an inode, and completes its changes, it must call `iunlock` (line 5085) to release the lock. This function simply clears the busy lock and wakes up any processes blocking on the lock. At this point after unlocking the inode, the changes made to the inode are in the inode cache, but not on disk. To flush to the disk, one needs to call `iupdate` (line 4979). Finally, to release C pointer reference, a process must call `iput` (line 5108). Note that one must always call `iunlock` before `iput`, because if the reference is decremented and goes to zero, the inode may no longer be valid in memory.

- When can a file be deleted from disk? In other words, when is an inode freed on disk? When it has no links or C references in memory. So the best place to check this condition is in the `iput` function. When releasing the last memory reference to an inode, `iput` checks if the link count is also zero. If yes, it proceeds to clean up the file. First, it locks the inode by setting it to busy, so that the in-memory inode is not reused due to a zero reference count. Then, it releases all data blocks of the the file by calling `itrunc` (line 5206), clears all state of the inode, and updates it on disk. This operation frees up the inode on disk. Next, it clears the flags of the in-memory inode, freeing it up as well. Since the reference count of the in-memory inode is zero, it will be reused to hold another inode later. Note how `iput` releases and reacquires its lock when it is about to do disk operations, so as to not sleep with a spinlock.

- Notice how all the above operations that change the inode cache must lock `icache.lock`, because multiple processes may be accessing the cache concurrently. However, when changing blocks on disk, the buffer cache handles locking by giving access to one process at a time in the function `bread`, and the file system code does not have to worry about locking the buffers itself.

- Finally, the functions `readi` and `writei` are used to read and write data at a certain offset in a file. Given the inode, the data block of the file corresponding to the offset is located, after which the data is read/written. The function `bmap` (line 5160) returns the block number of the n-th block of an inode, by traversing the direct and indirect blocks. If the n-th block doesn't exist, it allocates one new block, stores its address in the inode, and returns it. This function is used by the `readi` and `writei` functions.

5. Directories and pathnames

- A directory entry (line 3950) consists of a file name and an inode number. A directory is also a special kind of file (with its own inode), whose content is these directory entries written sequentially. Sheets 53–55 hold all the directory-related functions.

- The function `dirlookup` (line 5361) looks over each entry in a directory (by reading the directory "file") to search for a given file name. This function returns only an unlocked inode of the file it is looking up, so that no deadlocks possible, e.g., look up for "." entry. The function `dirlink` (line 5402) adds a new directory entry. It checks that the file name doesn't exist, finds empty slot (inode number 0) in the directory, and updates it.

- Pathname lookup involves several calls to `dirlookup`, one for each element in the path name. Pathname lookup happens via two functions, `namei` and `nameiparent`. The former returns the inode of the last element of the path, and the latter stops one level up to return the inode of the parent directory. Both these functions call the function `namex` to do the work. The function `namex` starts with getting a pointer to the inode of the directory where traversal must begin (root or current directory). For every directory/file in the path name, the function calls `skipelem` (line 5465) to extract the name in the path string, and performs `dirlookup` on the parent directory for the name. It proceeds in this manner till the entire name has been parsed.

- Note: when calling `dirlookup` on a directory, the directory inode is locked and updated from disk, while the inode of the name looked up is not locked. Only a memory reference from `iget` is returned, and the caller of `dirlookup` must lock the inode if needed. This implementation ensures that only one inode lock (of the directory) is held at a time, avoiding the possibility of deadlocks.

6. File descriptors

- Every open file in xv6 is represented by a `struct file` (line 4000), which is simply a wrapper around inode/pipe, along with read/write offsets and reference counts. The global open file table `ftable` (line 5611-5615) is an array of file structures, protected by a global `ftable.lock`. The per-process file table in `struct proc` stores pointers to the `struct file` in the global file table. Sheets 56–58 contain the code handling `struct file`.

- The functions `filealloc`, `filedup`, and `fileclose` manipulate the global filetable. When an open file is being closed, and its reference count is zero, then the in-memory reference to the inode is also released via `iput`. Note how the global file table lock is released before calling `iput`, in order not to hold the lock and perform disk operations.

- The functions `fileread` and `filewrite` call the corresponding read/write functions on the pipe or inode, depending on the type of object the file descriptor is referring to. Note how the fileread/filewrite functions first lock the inode, so that its content is in sync with disk, before calling `readi/writei`. The `struct file` is one place where a long term pointer to an inode returned from `iget` is stored. The inode is only locked when the file has to be used for reading, writing, and so on. The function `filewrite` also breaks a long write down into smaller transactions, so that any one transaction does not exceed the maximum number of blocks that can be written per transaction.

7. System calls

- Once a `struct file` is allocated in the global table, a pointer to it is also stored in the per-process file table using the `fdalloc` function (line 5838). Note that several per-process file descriptor entries (in one process or across processes) can point to the same struct file, as indicated by the reference count in `struct file`. The integer file descriptor returned after file open is an index in the per-process table of pointers, which can be used to get a pointer to the `struct file` in the global table. All file-system related system calls use the file descriptor as an argument to refer to the file.

- The file system related system calls are all implemented in sheets 58-62. Some of them are fairly straightforward: they extract the arguments from the stack, and call the appropriate functions of the lower layers of the file system stack. For example, the read/write system calls extract the arguments (file descriptor, buffer, and the number of bytes) and call the fileread/filewrite functions on the file descriptor. It might be instructive to trace through the read/write system calls through all the layers down to the device driver, to fully understand the file system stack.

- The `link` system call (line 5913) creates a pointer (directory entry) from a new directory location to an old existing file (inode). If there were no transactions, the implementation of the link system call would have resulted in inconsistencies in the system in case of system crashes. For example, a crash that occurs after the old file's link count has been updated but before the new directory entry is updated would leave a file with one extra link, causing it to be never garbage cleaned. However, using transactions avoids all such issues. The `unlink` system call (line 6001) removes a path from a file system, by erasing a trace of it from the parent directory.

- The function `create` (line 6057) creates a new file (inode) and adds a link from the parent directory to the new file. If the file already exists, the function simply returns the file's inode. If the file doesn't exist, it allocates and updates a new inode on disk. Then it adds a link from the parent to the child. If the new file being created is a directory, the "." and ".." entries are also created. The function finally returns the (locked) inode of the newly created file. Note that `create` holds two locks on the parent and child inodes, but there is no possibility of deadlock as the child inode has just been allocated.

- The `open` system call (line 6101) calls the `create` function above to get an inode (either a new one or an existing one, depending on what option is provided). It then allocates a global file table entry and a per-process entry that point to the inode, and returns the file descriptor. The function `create` is also used to create a directory (line 6151) or a device (line 6167).

- The `chdir` system call simply releases the reference of the old working directory, and adds a reference to the new one.

- To create a pipe, the function `pipealloc` (line 6471) allocates a new pipe structure. A pipe has the buffer to hold data, and counters for bytes read/written. Two `struct file` entries are created that both point to this pipe, so that read/write operations on these files go to the pipe. Finally file descriptors for these file table entries are also allocated and returned.