

Today's goal: Synthesis

- + Walk through two system calls in some detail
 - + Open and read
- ✤ Too much code to cover all FS system calls

Quick review: dentry

- + What purpose does a dentry serve?
 - * Essentially maps a path name to an inode
- More in 2 slides on how to find a dentry
- + Dentries are cached in memory
 - Only "recently" accessed parts of a directory are in memory; others may need to be read from disk
 - + Dentries can be freed to reclaim memory (like pages)

Dentry caching

- + 3 Cases for a dentry:
 - In memory (exists)
 - Not in memory (doesn't exist)
 - Not in memory (on disk/evicted for space or never used)
- ✤ How to distinguish last 2 cases?
 - * Case 2 can generate a lot of needless disk traffic
 - + "Negative dentry" Dentry with a NULL inode pointer

Dentry tracking

- * Dentries are stored in four data structures:
 - * A hash table (for quick lookup)
 - * A LRU list (for freeing cache space wisely)
 - + A child list of subdirectories (mainly for freeing)
 - * An alias list (to do reverse mapping of inode -> dentries)
 - * Recall that many directories can map one inode

Open summary

- * Key kernel tasks:
 - * Map a human-readable path name to an inode
 - Check access permissions, from / to the file
 Possibly create or truncate the file (O_CREAT,
 - O_TRUNC)
 - Create a file descriptor

Open arguments

- + int open(const char *path, int flags, int mode);
- + Path: file name
- + Flags: many (see manual page), include read/write perms
- Mode: If a file is created, what permissions should it have? (e.g., 0755)
- Return value: File handle index (>= 0 on success)
 - ✤ Or (0 –errno) on failure

Absolute vs. Relative Paths

- * Each process has a current root and working directory
 - Stored in current->fs-> (fs, pwd---respectively)
 - + Specifically, these are dentry pointers (not strings)
- * Note that these are shared by threads
- Why have a current root directory?
 - + Some programs are 'chroot jailed' and should not be able to access anything outside of the directory

More on paths

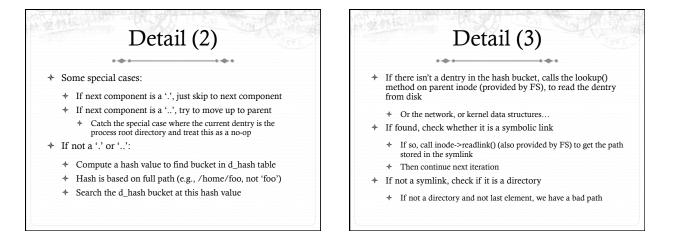
- * An absolute path starts with the '/' character
 - * E.g., /home/porter/foo.txt, /lib/libc.so
- A relative path starts with anything else: ÷
 - + E.g., vfs.pptx, ../../etc/apache2.conf
- + First character dictates where in the dcache to start searching for a path

Search ✤ Executes in a loop, starting with the root directory or the current working directory ✤ Treats '/' character in the path as a component delimiter ✤ Each iteration looks up part of the path

- * E.g., '/home/porter/foo' would look up 'home', 'porter', then 'foo', starting at /

Detail (iteration 1)

- ✤ For current dentry (/), dereference the inode
- + Check access permission (recall, mode is stored in inode)
 - Use a permission() function pointer associated with the + inode - can be overridden by a security module (such as SeLinux, or AppArmor), or the file system
- + If ok, look at next path component (/home)



Iteration 2

- Check permission in /home
- * Hash /home/porter, find dentry
- Confirm not '.', '..', or a symlink
- ✤ Confirm is a directory
- ✤ Recur with dentry/inode for /home/porter, search for foo

Symlink problems

•-----•••

- * What if /home/porter/foo is a symlink to 'foo'?
- Kernel gets in an infinite loop
- * Can be more subtle:
 - ✤ foo -> bar
 - ✤ bar -> baz
 - + baz -> foo

Preventing infinite recursion

- ✤ More simple heuristics
- * If more than 40 symlinks resolved, quit with –ELOOP
- If more than 6 symlinks resolved in a row without a nonsymlink inode, quit with -ELOOP
- + Maybe add some special logic for obvious self-references
- * Can prevent execution of a legitimate 41 symlink path
 - Generally considered reasonable

Back to open()

✤ Key tasks:

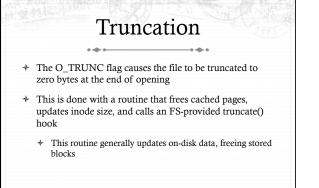
- * Map a human-readable path name to an inode
- * Check access permissions, from / to the file
- Possibly create or truncate the file (O_CREAT, O_TRUNC)
- * Create a file descriptor
- * We've seen how steps 1 and 2 are done

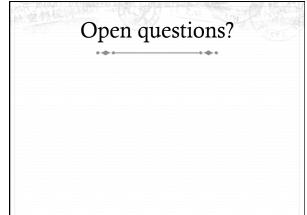
Creation

- + Handled as part of search; treat last item specially
- + Usually, if an item isn't found, search returns an error
- * If last item (foo) exists and O_EXCL flag set, fail
- If O_EXCL is not set, return existing dentry
- ✤ If it does not exist, call fs create method to make a new inode and dentry
 - * This is then returned

File descriptors

- User-level file descriptors are an index into a processlocal table of struct files
- + A struct file stores a dentry pointer, an offset into the file, and caches the access mode (read/write/both)
 - * The table also tracks which entries are valid
- Open marks a free table entry as 'in use'
 - * If full, create a new table 2x the size and copy old one
 - * Allocates a new file struct and puts a pointer in table





Now on to read

* int read(int fd, void *buf, size_t bytes);

--

- ✤ fd: File descriptor index
- ✤ buf: Buffer kernel writes the read data into
- + bytes: Number of bytes requested
- * Returns: bytes read (if >= 0), or -errno

Simple steps

- + Translate int fd to a struct file (if valid)
- * Check cached permissions in the file
- Increase reference count
- * Validate that sizeof(buf) >= bytes requested
- And that buf is a valid address
- Do read() routine associated with file (FS-specific)
- * Drop refcount, return bytes read

Hard part: Getting data

- In addition to an offset, the file structure caches a pointer to the address space associated with the file
- * Recall: this includes the radix tree of in-memory pages
- $\ensuremath{\bigstar}$ Search the radix tree for the appropriate page of data
- + If not found, or PG_uptodate flag not set, re-read from disk
- If found, copy into the user buffer (up to inode->i_size)

Requesting a page read

- ✤ First, the page must be locked
 - * Atomically set a lock bit in the page descriptor
 - * If this fails, the process sleeps until page is unlocked
- Once the page is locked, double-check that no one else has re-read from disk before locking the page
 - + Also, check that no one has freed the page while we were waiting (by changing the mapping field)
- Invoke the address_space->readpage() method (set by FS)

Generic readpage

- ✤ Recall that most disk blocks are 512 bytes, yet pages are 4k
 - + Block size stored in inode (blkbits)
- Each file system provides a get_block() routine that gives the logical block number on disk
- Check for edge cases (like a sparse file with missing blocks on disk)

More readpage

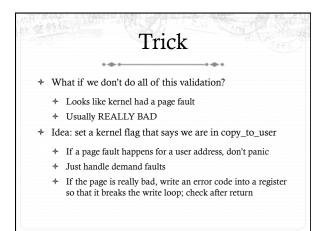
- If the blocks are contiguous on disk, read entire page as a batch
- + If not, read each block one at a time
- These block requests are sent to the backing device I/O scheduler (recall lecture on I/O schedulers)

After readpage

- * Mark the page accessed (for LRU reclaiming)
- Unlock the page
- Then copy the data, update file access time, advance file offset, etc.

Copying data to user

- * Kernel needs to be sure that buffer is a valid address
- + How to do it?
 - * Can walk appropriate page table entries
- * What could go wrong?
 - * Concurrent munmap from another thread
 - * Page might be lazy allocated by kernel



Benefits

- This trick actually speeds up the common case (buf is ok)
- * Avoids complexity of handling weird race conditions
- * Still need to be sure that buf address isn't in the kernel

