CS 5600
Computer Systems

Project 4: File System in Pintos

File System in Pintos

- Pintos already implements a basic file system
  - Can create fixed size files in a single root directory
- But this system has limitations
  - No support for nested directories
  - No support for files that grow in size
  - No caching or preemptive reading

Your Goals

1. Implement indexed files
   - Files should begin life as a single sector and grow dynamically as necessary
   - Processes should be able to seek and write past the end of a file
   - Requires heavily modifying Pintos' inodes
2. Implement nested directories
   - You will need to implement new system calls to manipulate directories
   - chdir(), mkdir(), readdir(), isdir()
   - Inode management: inumber() → get the inode of a file or directory

Your Goals (cont.)

3. Implement a buffer cache
   - Up to 64 sectors of disk data should be buffered in RAM
   - Implement a write-back cache
   - Cache must be periodically flushed to disk
   - How to handle eviction?
4. Carefully synchronize file operations
   - Accesses to independent files/directories should not block each other
   - Concurrent reading/writing of a single file needs to be handled carefully

What Pintos Does For You

- Basic disk management
  - Read/write access to sectors
  - Basic management of free space
- You've already implemented file descriptors and most of the file system API ;)

Inodes in Pintos

- filesys/inode.c

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

Directories in Pintos

- filesys/directory.c

- Implements a single root directory
  - i.e. no subdirectories
- Must be overhauled to allow a directory to contain other directories
  - e.g. subdirectories

Key Challenges

- Choosing the right data structures
  - How do you encode directory and file information on disk?
  - How do you keep track of the locations of dynamically allocated file blocks
- Properly managing your cache
  - Implementing performant cache eviction is tricky
  - Write-back cache must be periodically flushed
- Implementing correct and performant synchronization

More Key Challenges

- Each process needs to have an associated working directory
  - Necessary for resolving relative file accesses
    - E.g. open("../file.txt") or open("./my_thing")
  - Used by the *pwd* program

Modified Files

- filesys/Make.vars 6
- filesys/cache.c 473 # new file!
- filesys/cache.h 23 # new file!
- filesys/directory.c 99
- filesys/directory.h 3
- filesys/file.c 4
- filesys/filesys.c 194
- filesys/filesys.h 5
- filesys/free-map.c 45
- filesys/free-map.h 4
- filesys/fsutil.c 8
- filesys/inode.c 444
- filesys/inode.h 11
- userprog/process.c 12
- userprog/syscall.c 37
- 15+ files changed, 1368 insertions(+), 286 deletions(-)

This Project Is the Biggest

- The reference solution for Project 4 includes way more lines of code than any other project thus far

Start early!

Dependency on older Projects

- Project 4 can built on top of Project 2 or Project 3
- If you build on top of Project 3, requires having a rock-solid VM implementation

DUE: December 5
11:59:59PM PST

NO EXTENSIONS FOR THIS ONE!