

This is an assignment in which you will implement the System V File System. The goal of the assignment is to reinforce the file system concepts, and maintain both in-memory and on-disk state related to a file system. In doing so, you will implement the interface for much of the high-level functionality associated with a file system.

In addition, this assignment relies heavily on “virtualization” in C programming. This will give you a sense of how we can implement and represent virtualization in high-level languages in a lower-level language that does not have native support for classes, virtual functions, etc... We use virtualization in this assignment to highlight that `objs` do not necessarily need to implement `vnodes`, `vnodes` do not necessarily need to implement `svfs_nodes`, and so on! This modularity allows the OS kernel to implement *any* file system and/or support *multiple* file systems if necessary (e.g., to make use of virtual machines)! While this assignment does not explicitly ask you to implement support for multiple file systems, hopefully you will gain an appreciation for how we could use this infrastructure to implement such a behavior.

Collaboration Policy

This assignment is intended to be implemented as a *pair-programming* assignment (often in groups of 3)! That means that you and your partner are working on the assignment **together**.

After the initial programming phase of the assignment, you and your partner will participate in a *code review* of another group’s submission. Be sure to review the code reviewing standards from Homework 0 submit your code review on Gradescope!

Submission

You are asked to submit three deliverables for this assignment: ① the programming component, ② your code review for your partnered group, and ③ a reflection to the review of your code. In addition, in-class participation is required on **November 2nd** to both give and receive feedback for full credit.

Getting Started

This assignment is designed to run on the course virtual machine (e.g., `ssh stum2025@itbdcv-lnx04p.campus.pomona.edu`). The stencil for assignments in this class assume particular compiler versions and processor features, so using the VM will make your life easier.

Download the source for this assignment to get started! You can find all of the tasks that you will need to implement by running `grep "TODO: implement me!" -r .` from the “fs-stencil” repository. In the repository, you will find a `Makefile` to help you build the various sources for each part of this assignment.

Deadlines

Section	Deadline	Submission URL
FS Maker	Oct 16	(suggested deadline)
Programming Component	Oct 26 (EOD)	HW3 Programming
Code Review	Nov 2 (before class)	HW3 Code Review
Code Review Reflection	Nov 2 (EOD)	HW3 Code Review Reflection
Post-Review Resubmission	Nov 6 (EOD)	HW3 Programming (late due date)

1 Before You Begin

1.1 System V File System (CS134 version)

In this assignment, you will be implementing a version of the System V File System. In particular, the assignment starts with a blank disk image (a file of null bytes), and asks you to transform the blank image into a usable file system. To do so, your file system will follow the following specs:

- block size: 4096 bytes
- num boot blocks: 0
- num superblocks: 1
- num inodes: 32
- num data blocks: 128

In implementing the file system, there is not enough space to maintain the entire free list of data blocks in the superblock. In fact, the superblock only has space to track 30 unused data blocks. This means that the final entry should be a reference to a data block that indicates the remaining free data blocks.

Additionally, implementing the list of free inodes is done in the superblock by maintaining a reference to the first free inode in the inode segment of the disk. Then, inodes are implemented to either refer to the next free inode (if the inode is free) or the size of the file that the inode represents (if the inode describes a file). This means that the inode list is implemented *in the free inodes themselves*. This means that the mode of the inode carries a lot of importance! It tells us how to interpret various fields within the inode struct.

1.2 System Overview

Our OS kernel will *not* directly implement the mounting procedure to load the file system from the disk into its state. To account for this, we instead leverage a `config.cfg` file that tells the OS kernel which file to use as its simulated disk. To create a version of this file, run `make empty-iso` to create a file with the right number of blocks to implement your file system. The `sys/disk/disk.{h,c}` files will implement a synchronous interface to read/write from disk¹, and have been provided for you.

To reason about data that lives on disk and in memory, you have been provided with a `mobj` interface that will allow you to reason about data that lives on disk in memory. The interfaces have been provided to you in the `include/mobj.h` file, and you are asked to implement several of the functions in `src/mobj.c`. In general, we can think of `mobj` structs as wrappers for `pframes`, which each have a “buffer” of 4096 bytes to keep the bytes from the frame on disk in “memory.”

On top of the `mobj` interface, the file system implements `vnodes`, the fundamental data structure to represent files in a file system agnostic way. There is not much for you to implement in `vnode`, but take a look at the layout of the structure in the `include/vnode.h` file to see the various fields of a `vnode` object.

Finally, to make use of `vnodes` from a System V perspective we implement `svfs_nodes` in `include/svfs.h`. These structs wrap the `vnode` struct with more awareness of the specific

¹In practice, this would be an asynchronous procedure interfacing with a disk driver that uses a disk protocol (e.g., SATA) to send bytes to the appropriate device.

details pertaining to the System V file system (e.g., which functions are allowed for files versus directories, how to interpret various disk block type layouts, etc...). Note that, even at this level of abstraction, there are helper functions to access almost every functionality across the stack to implement the necessary procedures as it relates to a file in disk, in memory, or a non-existent file to create! For this reason, it is useful to take a moment to orient yourself in the project before diving into the details of the various components.

2 File System Maker

In the first part of the assignment, your task will be to write a script that builds a disk image with a file system on it. In doing so, you will be able to construct a file system that your OS kernel will be able to load and use during its execution. Note, *there are no requirements on the script format!* That is you may wish to write this script in C, C++, Java, Python, or any other programming language! However you choose to implement it, please create a file called `fs-maker.{ext}` with the appropriate file extension for your script.

To implement the file system maker, be sure to look through the expected format of how the various blocks will be interpreted in the `svfs.h` header. This will dictate how bytes should be arranged on your disk so that the file system is usable.

2.1 Superblock Initialization

The index of the superblock on our disk is known in advance. This will make it easy to validate the superblock in the `svfs.c` source when we fetch the block from disk. Beyond this, it is your task to implement the various fields on disk. Note, you merely need to write the *values* of these fields (with the size of the types in mind) to the disk. When we read the bytes and interpret them as a `svfs_superblock_t` these bytes will be interpreted as the appropriate fields.

Note, part of the task of implementing the superblock is also implementing the free list. The superblock has space for some (not all) entries in the free list. All data blocks on the disk should be accessible via the free list.

2.2 Inode Initialization

Much like how you initialize the superblock, you will also initialize the inodes on the disk. To do so, be sure to consider what the various fields of the `svfs_inode_t` struct would be set to in the event that the inode is not yet initialized. Note, to do so the `inode` uses a *union* to represent either the next free block or the size of the file, depending on the state of the inode. You will need to take this into account when writing these values to disk. Much like the free list of data blocks, be sure that your inodes are appropriately implementing the inode free list.

2.3 Base File System

In this assignment, you are not asked to “mount” the file system implemented on this disk. As such, your file system should be an exact replica of the file system that you will construct on boot. In particular, you should construct a single directory representing the file at the file system root in your disk, which will occupy the first “free” inode and data block as a directory type (i.e., `/`).

Recall, directories have their first two entries reserved for the “.” and “..” directories. Whenever you create a new directory, make sure that these entries are set!

2.4 Testing

To demonstrate that your `fs-maker` is working as expected, please be sure to test for *at least* the following features:

1. The superblock format is created in such a way that, when read and interpreted as a superblock, the fields of struct make sense relative to your layout;
2. Free inodes refer to each other to construct the inode free list;

3. The free list of data blocks appropriately refers to the various data blocks in the system, and at least one data block is used to track the free blocks that do not fit in the superblock.

3 mobj

In this section, you are asked to implement the interface between the frames of the disk and pages in memory. To do so, you will implement the functions defined in `mobj.c` that allow for different interactions between the data as it lives in memory and on disk.

Note, in this version of the assignment, we are not placing the bytes of the file in the “address space” of the OS kernel nor the processes. Instead, we leverage the `pframe_t` struct (declared in `include/pframe.h`) which has a buffer of 4096 bytes to maintain the disk bytes in “memory.” However, because files may be large (e.g., several data blocks), our definition of a `mobj` tracks the various frames by maintaining these `pframes` in a list. The order of this list is important: the first index element of the list should be the first frame of the file, etc... to ensure that any operation on our `mobj` is performed across `pframes`.

In general, our file system will implement three primitive interactions with bytes in our file system: reading, writing, and flushing. Note, these operations may or may not be aligned to the pages/frames/disk blocks. It is your task to implement each of these functions (considering these cases!). Their high-level behaviors are detailed below with implementation details and hints provided in the scaffolding.

3.1 Read to mobj

When we read files, we need to access the data in the frames associated with the `mobj`. In general, however, we cannot assume that all of the frames of the file will necessarily be in the `mobj` at the time that the read is initiated. If this is the case, we read as many bytes as we can from the data that is currently in the in-memory `pframes`, and tell the caller how many bytes were read through the return value.

Note, if not all `pframes` are in the `mobj` at the time of the read, it is the job of the *caller* to direct the `mobj` to fetch more frames from disk to keep in the `mobj` list. The `mobj` itself will not necessarily know where these subsequent frames are, as at this level of abstraction we do not have knowledge of the disk layout. The caller will do this by calling the `mobj_get_pframe` function to load the `pframe` into memory from disk.

3.2 Writing to mobj and Flushing

Much like when reading files, we cannot necessarily guarantee that we will be able to complete all writes to a `mobj`. When performing a write, the `mobj` should update the copy of the data *in memory* (e.g., the bytes associated with the `pframe`).

The copy of the data on disk is only ever updated when there is an explicit call to `mobj_flush`. This utilizes the disk interface to move the data from the associated `pframes` to their disk version.

4 vnode

This section provides an overview of the `vnode` struct. Note, there are no implementation requirements on your end for this section, but understanding this section will help contextualize the implementation of the `svfs`-related structs and methods as well as the `namev` methods (which are `vnode` abstractions of the file system-specific function calls).

4.1 `vnode_ops_t`

When implementing a `vnode`, we need to describe all of the operations that could be implemented by the file system with respect to that `vnode`. It may be the case that the file system has other methods as well, but at a minimum any file system that builds on top of the `vnodes` needs to implement these functions. In particular, these functions are:

- `read`: read from a `vnode`
- `write`: write to a `vnode`
- `mknod`: create a new `vnode` for a regular file
- `mkdir`: create a new `vnode` for a directory
- `lookup`: if the `vnode` is a directory, find a file with the same name/`namelen` as the provided input
- `readdir`: get the “pos”-th entry from the `vnode` if the `vnode` is a directory
- `read_pframe`: read from a `pframe` in the `mobj` associated with the `vnode`
- `write_pframe`: write to a `pframe` in the `mobj` associated with the `vnode`
- `flush_pframe`: flush the `pframe` associated with this `vnode` to disk

Note, the way that we have declared these functions is merely as a struct of pointers. Because pointers are of a fixed size, there does not necessarily need to be a definition of these functions in the `vnode`, but whenever a `vnode` is created, there should be pointers to various sets of these functions (as appropriate). This is how we can implement *virtualization* in C despite not having access to classes.

4.2 The `vnode_t` struct

Once we have declared the functions that can be called on a `vnode`, we have the requisite information to describe the `vnode_t` struct. In the struct, we maintain a reference to: ① an implementation of the operations that were just described, ② a reference to the file system that the `vnode` belongs to, ③ the `mobj` associated with the `vnode`, ④ some metadata about the `vnode` (length and mode – where are these defined in the file system?), ⑤ a link to be maintained on any global list of `vnodes`, and ⑥ a reference to its parent (e.g., a `node` struct in the file system that uses the `vnode`).

In general, the `vnode` serves as a wrapper for file systems built on top of the `vnode` interface to use the implementations of these various functions. You may use the `src/vnode.c` code as an example of typical `vnode` usage. In `vnode_read_pframe`, for example, the `mobj` is used for its reference to its parent (e.g., the `vnode` that contains it) so that we can dereference its associated `vn_ops` and call the appropriate functionality.

Also of note in this example, you will see that we declare an instance of `mobj_ops` for the `vnode` in which `vnodes` can declare what it means to call the various `mobj` functionalities. In the context of “`read_pframe`”, for instance, we call the virtualized `vnode` function to read the `pframe` from the `vn_ops` (which will be implemented by the file system using the `vnodes`).

5 System V File System (SVFS)

In general, file systems can be thought of as collections of `vnodes`. The declaration of an abstract file system is declared in `include/vfs.h`. In this file, a file system is declared as being a root `vnode`, an abstract set of functions to both read `vnodes` and initialize the file system, and mechanisms to allocate and track the `vnodes` in the system. This will allow user-space software to reason about the file system tree implemented by the file system structure in terms of `vnodes`, which are agnostic to any particular implementation of a file system.

Beyond the abstract implementation of a file system, the System V File System needs to implement significant breadth of functionality. The `vnode` struct relies on the file system implementation to provide definitions of each of the `vnode_ops` so that the `vnode` can fully implement the `mobj_ops`. In general, we can think of the classes of operations that the System V File System implements in three groups.

5.1 File System Functions

In order to implement the abstract file system definition, the System V File System needs to implement a “`read_vnode`” function and “`fs_init`” function. These are used in `svfs.c` to define an instance of the `fs_ops` struct, where the particular definitions are declared as a variable called `svfs_fsops`. Note, when initializing the System V file system, fields to instantiate may be declared across the `svfs.h` and `vfs.h` headers.

5.2 Regular File Functions

In the same way that the `svfs_fsops` are used to provide a System V definition of the `fs_ops` struct, we can use a set of files that map to various System V functions to describe the legal operations that may be performed by regular files. In general, there are certain functions that *should not be allowed* for regular files. When this is the case, the pointer for these functions is defined as `NULL` in the `svfs_file_vops` struct definition.

5.3 Directory Functions

Much like with the regular files, directories also have certain operations that should and should not be allowed. These too are represented as references to functions defined in the System V file system or as `NULL`, depending on the set of legal behavior. In addition, the data associated with directories follows a well defined format (declared in `include/dirent.h`). That is, a directory can be thought of as a collection of `dirents`, and this data can be used to interpret this file type to implement directory-related functions.

Your Task in this section of this assignment is to implement the various functions declared in the `svfs.h` header to implement the System V File System. Note, for many of these functions, you will need to refer to various sub-functionalities of various components of the system stack. For instance, if you need to perform a disk operation, you may need to access the `mobj` associated with the file from the `svfs_...` function. A holistic understanding of the system will greatly benefit you towards this end.

5.4 `namev`

To implement certain generally virtualizable functionalities of a file system holistically (not just in terms of `vnodes`), the file system (`vfs.h`) declares certain functions that all file systems should

implement. These functions instantiate generalizable declarations of the `svfs_...` functions that you've implemented throughout this part of the assignment.

Your task in this section of the assignment also includes a generic implementation of these functions. In general, most of your functions should refer to implementations of similar functions in the System V file system without calling these functions directly. *Hint: think about what this means in terms of virtualization.*

6 Tests

When performing the code review (and testing the functionality of your own code), there are several tests that should implement.

1. You should put together a test script that launches an instance of the System V File System in which you can read a file from your *host* machine and write that file into your simulated file system;
2. You should put together a test script that launches an instance of the System V File System in which you can read a file from your *simulated* file system onto your host;
3. You should have a test in which a file created in your file system's memory may be flushed to the disk image such that the file is readable on subsequent runs of the same and/or other tests;
4. You should implement at least one other test as well!

7 Extensions

7.1 Loading Files into File System

Extend your FS Maker to create a disk image that is non-empty! This will allow you to boot your fully functional OS kernel with custom, accessible binaries that can be run!

7.2 File Deletion

As currently constructed, files are created in this file system and live throughout the lifetime of the system (in memory). Extend your implementation such that vnodes may be released when they are no longer in use. This will require a system of reference counting, as several processes may all have file descriptors to a file.

7.3 File System Awareness in the Process

Extend your process struct so that it maintains both: ① a reference to a vnode describing the current working directory, and ② a list of currently open file descriptors.

7.4 File Permissions

Extend your inode definition to ensure that files can only be read, written, and executed as appropriate by certain users.

7.5 mobj Cache

When implementing the mobj system, keep various mobjs in an in-memory cache. Infrequently accessed pages or mobjs may be kept on disk.

7.6 mobj In Memory As Opposed to Using “In-Struct Buffers”

As part of the OS kernel memory, reserve space for the mobj structs and allocate pages in the OS kernel memory for every page as opposed to keeping them in an in-memory buffer in the struct.

7.7 Clean Shutdown

When turning off your system, ensure that everything is backed up to disk!