CS350 Operating Systems Winter 2022

Final Exam

1 Introduction

There are two components to the CS350 Final Exam for Winter 2022. Read these instructions clearly.

Quiz 5 - Final Quiz - The quiz will be on Learn. It will be open for a 24 hour window. The quiz will have restrictions on the duration. The enforced time will be 40 minutes. The material covered in the quiz will be cumulative:

- Processes
- System Calls and Interrupts
- Threads
- Concurrency
- Synchronization
- Scheduling
- Virtual Memory - Hardware
- Virtual Memory - OS

There will be **more** focus on the material in the following sections ,which were not covered by previous quizzes:

- File I/O
- File Systems

The final quiz will be available on **Apr 13th 2022 starting at 12:01am until Apr 13th 11:59 pm.** You should see the availability of the quiz on Learn.

Linux `userspace` Programming Assignment - The final porgramming assignment is to implement a simplified version of the Unix File System, called **SimpleFS**. The progrmaming assignment will be in `C`. The final programming assignment is **due on April 20th at 5pm, no further extensions**. In this final exam programming poject specification, we will provide:

- explicit implementation requirements for the final programming assignment
- starter code - with a `Makefile`

- instructions for submitting the final programming assignment/project

Some general advice for this assignment:

- **Start early.** The instructions are detailed, but even debugging simple mistakes are time consuming. This holds doubly true if you are not especially familiar with `C`.
- **Compile often.** By checking whether the code compiles after small modifications, you will be able to pinpoint problems very quickly.

2 Implementation Requirements

In this assignment, there are three components:

Shell: The first component is a simple shell program that allows the user to perform operations on the SimpleFS. The shell supports built-in commands such as mount, create, read and write in the SimpleFS.

File System The second component takes the operation specified by the user through the shell and performs them on the SimpleFS disk image. This component has to organize the on-disk data structures and perform all the bookkeeping necessary to allow for persistent storage of data. To store the data, it will need to interact with the `disk emulator` via methods such as

- `disk_read`: which allows the file system to read from the disk image in 4096 byte blocks.
- `disk_write`: allows the file system to write to the disk image in 4096 byte blocks.

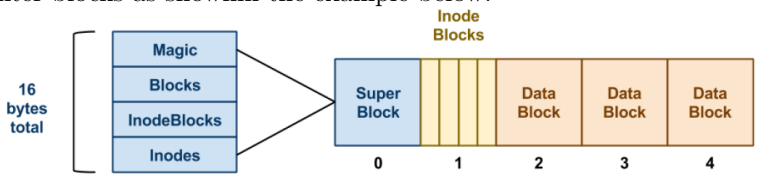
Disk Emulator: The third component emulates a disk by dividing a normal file (called a disk image) into 4096 byte blocks and only allows the SimpleFS to read and write in terms of the blocks. This emulator will persistently store the data to the disk image using the normal `open`, `read`, `write` system calls.

The `disk emulator` and the `shell` component is provided to you.
You have to complete the file system interface for the shell for this final programming project.

3 Simple File System (SimpleFS) Design

In this section, we describe the SimpleFS disk layout, with disk blocks are the common size of 4KB. The first block of the disk is the **superblock** that delineates the layout of the rest of the file system. A certain number of blocks

following the superblock contain **inode** data structures. Typically, ten percent of the total number of disk blocks are used as inode blocks. The remaining blocks in the filesystem are used as plain data blocks, and occasionally as indirect pointer blocks as shownin the example below:



In this example, we have a SimpleFS disk image that begins with a superblock . This superblock consists of four fields:

1. **Magic:** The first field is always the `MAGIC_NUMBER` or `0xf0f03410`. The format routine places this number in to the very first bytes of the superblock as a filesystem "signature". When the filesystem is mounted, the OS looks for this magic number. If it is correct, then the disk is assumed to contain a valid filesystem. If some other number is present, then the mount fails, perhaps because the disk is not formatted or contains some other kind of data.

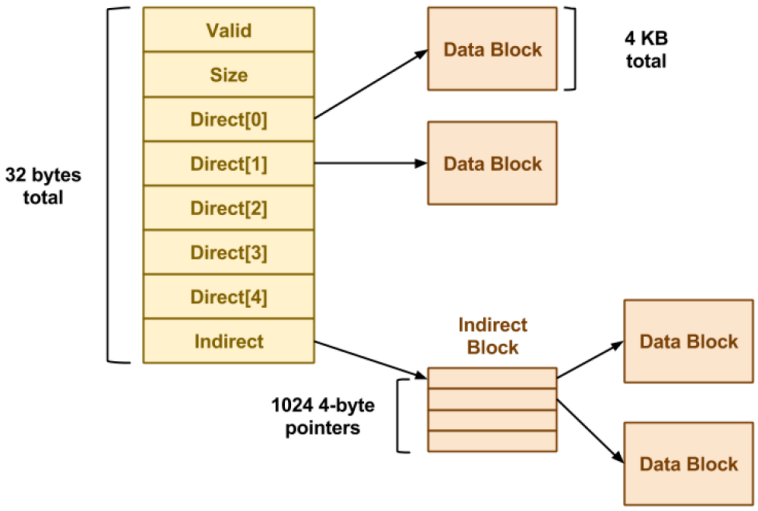
2. **Blocks:** The second field is the total number of blocks, which should be the same as the number of blocks on the disk.

3. **Inode Blocks:** : The third field is the number of blocks set aside for storing inodes. The format routine is responsible for choosing this value, which **should always be 10% of the Blocks**, rounding up, when necessary.

4. **Inode:** The fourth field is the total number of inodes in the inode blocks.

Note that the superblock data structure is quite small: only 16 bytes. The remainder of disk block zero is left unusued.

Each inode in SimpleFS looks like the following file:



Each field of the inode is a 4-byte (32-bit) integer. The `Valid` field is 1 if the inode is valid (i.e. has been created)and is 0, otherwise. The `Size` field contains the logical size of the inode data in bytes. There are 5 direct pointers todata blocks, and one pointer to an indirect data block. In this context, "pointer" simply means the number of a block where data may be found. A value of 0 may be used to indicate a *null* block pointer. Each inode occupies 32 bytes, so there are 128 inodes in each 4KB inode block.

Note that an indirect data block is just a big array of pointers to further data blocks. Each pointer is a 4-byte `int`, and each block is 4KB, so there are 1024 pointers per block. The data blocks are simply 4KB of raw data.

One thing missing in SimpleFS is the free block `bitmap`. A real filesystem would keep a free block bitmap on disk, recording one bit for each block that was available or in use. This bitmap would be consulted and updated every time the filesystem needed to add or remove a data block from an inode.

Since, the SimpleFS does not store this on-disk, you are required to keep a free block bitmap in memory. That is, there must be an array of integers, one for each block of the disk, noting whether the block is in use or available. When it is necessary to allocate a new block for a file, the system must scan through the array to locate an available block and allocate it. Similarly, when a block is freed, it must be marked in the bitmap.

Suppose that the user makes some changes to a SimpleFS filesystem, and then reboots the system (ie. restarts the shell). Without a free block bitmap, SimpleFS cannot tell which blocks are in use and which are free. Fortunately, this information can be recovered by scanning the disk. Each time that a SimpleFS filesystem is mounted, the system must build a new free block bitmap from scratch by scanning through all of the inodes and recording which blocks are in use.This is much like performing an `fsck` (file system check) every time the system boots.

SimpleFS looks much like the Unix file system. Each "file" is identified by an

integer called an `inumber`. The inumber is simply an index into the array of inode structures that starts in block one. When a file is created, SimpleFS chooses the first available inumber and returns it to the user. All further references to that file are made using the inumber.

For this assignment, you **are not required** to implement file and directory names.

4 Disk Emulator

We have provided you with a disk emulator to store your filesystem. This "disk" is actually stored as one big file in the filesystem, so that you can save data in a disk image and then retrieve it later. In addition, we will provide you with some sample disk images that you can experiment with to test your filesystem. Just like a real disk, the emulator only allows operations on entire disk blocks of 4 KB `BLOCK_SIZE` You cannot read or write any smaller unit than that. The primary challenge of building a filesystem is converting the user's requested operations on arbitrary amounts of data into operations on fixed block sizes.

The interface to the simulated disk is given in `disk.h`:

```
// disk.h: Disk emulator

#pragma once

#include <stdlib.h>
#include <stdbool.h>

#define BLOCK_SIZE 4096

typedef struct
{
    int FileDescriptor; // File descriptor of disk image
    size_t Blocks;      // Number of blocks in disk image
    size_t Reads;       // Number of reads performed
    size_t Writes;      // Number of writes performed
    size_t Mounts;      // Number of mounts
} Disk;

// Default constructor
Disk *new_disk();

// Destructor
// @param       disk pointer
void free_disk(Disk *disk);

// Open disk image
// @param       disk pointer
// @param       path        Path to disk image
// @param       nblocks     Number of blocks in disk image
void disk_open(Disk *disk, const char *path, size_t nblocks);

// Return size of disk (in terms of blocks)
// @param       disk pointer
size_t disk_size(Disk *disk);

// Return whether or not disk is mounted
// @param       disk pointer
bool disk_mounted(Disk *disk);

// Increment mounts
```

```
// @param       disk pointer
void disk_mount(Disk *disk);

// Decrement mounts
// @param       disk pointer
void disk_unmount(Disk *disk);

// Check parameters
// @param       disk pointer
// @param       blocknum    Block to operate on
// @param       data        Buffer to operate on
void disk_sanity_check(Disk *disk, int blocknum, char *data);

// Read block from disk
// @param       disk pointer
// @param       blocknum    Block to read from
// @param       data        Buffer to read into
void disk_read(Disk *disk, int blocknum, char *data);

// Write block to disk
// @param       disk pointer
// @param       blocknum    Block to write to
// @param       data        Buffer to write from
void disk_write(Disk *disk, int blocknum, char *data);
```

Before performing any sort of operation on the disk, you must call `disk_open` and specify a disk image for storing the disk data, and the number of blocks in the simulated disk. If this function is called on a disk image that already exists, the contained data will not be changed. When you are done using the disk, the destructor will automatcally release the file.

Make sure that your shell always opens the disk image first.

Once the disk is open, you may call `disk_size` to discover the number of blocks on the disk. As the names suggest, `disk_read` and `disk_write` read and write one block of data on the disk. For example:

```
Block block;
disk_read(disk, 0, block.Data);
```

Notice that the second argument is a block number, so a call to `disk_read(disk, 0, data)` reads the first 4KB of `data` on the `disk`, and `disk_read(disk, 1,data)` reads the next 4KB block of data on the disk. Every time that you invoke a read or a write, you must ensure that data points to a full 4KB of memory.

Additionally, you can register and unregister a disk as mounted by calling the `disk_mount` and `disk_unmount` functions respectively. The `disk_mounted` function returns whether or not the disk has been registerd as mounted.

Note that the disk has a few programming conveniences that a real disk would not. A real disk is rather finicky – if you send it invalid commands, it will likely crash the system or behave in other strange ways. This simulated disk is more"helpful." If you send it an invalid command, it will halt the program with an error message. For example, if you attempt to read or write a disk block that does not exist, it will throw an exception.

5 Implementing the SimpleFS

Using the disk emulator to build a working file system. We have provided the interface to the filesystem and with stub code in `fs.h` as shown below:

```
void fs_debug(Disk *disk);
bool fs_format(Disk *disk);

FileSystem *new_fs();
void free_fs(FileSystem *fs);

bool fs_mount(FileSystem *fs, Disk *disk);

ssize_t fs_create(FileSystem *fs);
bool fs_remove(FileSystem *fs, size_t inumber);
ssize_t fs_stat(FileSystem *fs, size_t inumber);

ssize_t fs_read(FileSystem *fs, size_t inumber,
                        char *data, size_t length, size_t offset);

ssize_t fs_write(FileSystem *fs, size_t inumber,
                        char *data, size_t length, size_t offset);
```

You are required to implement the functions as follows:

`fs_debug(Disk *disk):` This function scans a mounted filesystem and reports on how the inodes and blocks are organized. Your output from this method should be similar to the following:

```
% ./sfssh ../marking/data/image.5 5
sfs> debug
SuperBlock:
    magic number is valid
    5 blocks
    1 inode blocks
    128 inodes
Inode 1:
    size: 965 bytes
    direct blocks: 2
sfs>
```

`fs_format(Disk *disk):` This function creates a new filesystem on the disk, destroying any data already present. It should set aside ten percent of the blocks for inodes, clear the inode table, and write the superblock. It must return true on success, false otherwise. **Note: formatting a filesystem does not cause it to be mounted. Also, an attempt to format an already-mounted disk should do nothing and return failure.**

`fs_mount(FileSystem *fs, Disk *disk):` This function examines the disk for a filesystem. If one is present, read the superblock, build a free block bitmap, and prepare the filesystem for use. Return true on success, false otherwise. **Note: a successful mount is a pre-requisite for the remaining filesystem calls.**

`fs_create(FileSystem *fs)`: This function creates a new inode of zero length. On success, return the `inumber`. Otherwise, return -1 to signal failure.

`fs_remove(FileSystem *fs, size_t inumber)`: This function removes the inode indicated by the `inumber`. It should release all data and indirect blocks assigned to this inode and mark them as free in the free block map. On success, it returns true, false, otherwise.

`fs_stat(FileSystem *fs, size_t inumber)`: This method returns the logical size of the given `inumber`, in bytes. **Note: that `zero(0)` is a valid logical size for an inode.**On failure, it returns -1

`fs_read`: With arguments

- `FileSystem *fs`,
- `size_t inumber`,
- `char *data`,
- `size_t length`,
- `size_t offset`

This function reads data from a valid inode. It then copies `length` bytes from the data blocks of the inode into the `data` pointer, starting at `offset` in the inode. It should return the total number of bytes read. If the given `inumber` is invalid, or any other error is encountered, the function returns -1.

Note: the number of bytes actually read could be smaller than the number of bytes requested, perhaps if the end of the inode is reached.

`fs_write`: With arguments

- `FileSystem *fs`,
- `size_t inumber`,
- `char *data`,
- `size_t length`,
- `size_t offset`

This function writes data to a valid inode by copying `length` bytes from the pointer data into the data blocks of the inode starting at `offset` bytes. It will allocate any necessary direct and indirect blocks in the process. It returns the number of bytes actually written. If the given `inumber` is invalid, or any other error is encountered, return -1. Note: the number of bytes actually written could be smaller than the number of bytes request, perhaps if the disk becomes full.

It's quite likely that the File System struct will need additional internal member variables in order to keep track of the currently mounted filesystem. For example, you will certainly need a variable to keep track of the current free block bitmap, and perhaps other items as well. Feel free to modify the `fs.h` to include additional variables, structures or functions.

5.1 Implementation Notes:

Your job is to implement SimpleFS as described above by filling in the implementation of `fs.c`. We have already created some sample data structures to get you started. These can be found in `fs.h`. To begin with, we have defined a number of common constants that you will use. Most of these should be self explanatory:

```
#define MAGIC_NUMBER 0xf0f03410
#define INODES_PER_BLOCK 128
#define POINTERS_PER_INODE 5
#define POINTERS_PER_BLOCK 1024
```

Note that `POINTERS_PER_INODE` is the number of direct pointers in each inode structure, while `POINTERS_PER_BLOCK` is the number of pointers to be found in an indirect block. The superblock and inode structures are easily translated from the pictures above:

```
typedef struct
{                           // Superblock structure
    uint32_t MagicNumber; // File system magic number
    uint32_t Blocks;       // Number of blocks in file system
    uint32_t InodeBlocks;  // Number of blocks reserved for inodes
    uint32_t Inodes;       // Number of inodes in file system
} SuperBlock;

typedef struct
{
    uint32_t Valid;                       // Whether or not inode is valid
    uint32_t Size;                       // Size of file
    uint32_t Direct[POINTERS_PER_INODE];  // Direct pointers
    uint32_t Indirect;                    // Indirect pointer
} Inode;
```

Note carefully that many inodes can fit in one disk block. A 4KB chunk of memory containing 128 inodes would look like this:

```
Inode Inodes[INODES_PER_BLOCK];
```

Each indirect block is just a big array of 1024 integers, each pointing to another disk block. So, a 4KB chunk ofmemory corresponding to an indirect block would look liks this:

```
uint32_t
Pointers[POINTERS_PER_BLOCK];
```

Finally, each data block is just raw binary data used to store the partial contents of a file. A data block can be defined as an array of 4096 bytes:

```
char Data[BLOCK_SIZE];
```

Because a raw 4 KB disk block can be used to represent four different kinds of data: a superblock, a block of 128 inodes, an indirect pointer block, or a plain data block, we can declare a union of each of our four different data types. A union looks like a struct, but forces all of its elements to share the same memory space. You can think of a union as several different types, all overlaid on top of each other:

```
typedef union
{
    SuperBlock Super;                       // Superblock
    Inode Inodes[INODES_PER_BLOCK];         // Inode block
    uint32_t Pointers[POINTERS_PER_BLOCK];  // Pointer block
    char Data[BLOCK_SIZE];                  // Data block
} Block;
```

Note that the size of an Block union will be exactly 4KB : the size of the largest members of the union. To declare a Block variable:

```
Block block;
```

Now, we may use `disk_read()` to load in the raw data from block zero. We give`disk_read()` the variableblock.data, which looks like an array of characters: `disk_read(disk, 0, block.Data);` But, we may interpret that data as if it were a struct superblock by accessing the super part of the union. For example, to extract the magic number of the super block, we might do this:

```
x = block.Super.MagicNumber;
```

On the other hand, suppose that we wanted to load disk block 59, assume that it is an indirect block, and then examine the 4th pointer. Again, we would use `disk_read)` to load the raw data:

```
disk_read(disk, 59, block.Data);
```

But then use the pointer part of the union like this:

```
x = block.Pointers[4];
```

The union offers a convenient way of viewing the same data from multiple perspectives. When we load data from the disk, it is just a 4 KB raw chunk of data (`block.Data`). But, once loaded, the filesystem layer knows that this data has some structure. The filesystem layer can view the same data from another perspective by choosing another field in the union.

5.2 General Advice

1. Implement the functions roughly in order presented in `fs.h`. We have deliberately presented the functions of the filesystem interface in order of difficulty. Implement `fs_debug`, `fs_format`, and `fs_mount` first. Make sure that you are able to access the sample disk images provided. Then, perform creation and deletion of inodes without worrying abou tdata blocks. Implement reading and test again with disk images. If everything else is working, then attempt `fs_write`.

2. Divide and conquer. Work hard to factor out common actions into simple functions. This will dramatically simplify your code. For example, you will often need to find and store individual inode structures by number. This involves a fiddly little computation to transform an `inumber` into a block number, and so forth. So, make two little functions to do just that:

   ```
   bool find_inode(FileSystem *fs, size_t inumber, Inode *inode)

   bool store_inode(FileSystem *fs, size_t inumber, Inode *inode)
   ```

   Now, everywhere that you need to find or store an inode structure, you can call these functions. You may also wish to have functions that help you manage and search the free block map:

   ```
   void initialize_free_blocks();
   ssize_t fs_allocate_block(FileSystem *fs)
   ```

   Anytime that you find yourself writing very similar code over and over again, factor it out into a smaller function.

3. Test boundary conditions. We will certainly test your code by probing its boundaries. Make sure that you test and fix boundary conditions before handing in. For example, what happens if `fs_create` discovers that the inode table is full? It should cleanly return with an error code. It certainly should not crash the program or mangle the disk! Think critically about other possible boundary conditions such as the end of a file or a full disk.

6 The Shell and its Built-in Commands

The shell we have provided for you is an interactive loop that

- repeatedly prints the prompt "sfs> "
- parses the input
- executes the command specified on that line of input
- waits for the command to finish

This is repeated until the user types `exit` or `quit`.
In this project, the shell has the following built-in commands:

`help` - lists all the build-in SimpleFS commands that are supported by your shell.

`format` - format a new simpleFS

`mount` - mounting a SimpleFS

`debug` - print debugging information about the SimpleFS

`create` - create an `inode`

`remove` - remove an `inode`

`cat` - output information from an `inode`

`stat` - give stats about an `inode`

`copyin` - copying data in from the SimpleFS

`copyout` - copy data out to the SimpleFS

`help` - list the built-in commands provided in `sfs_shell`

`exit` - terminate the `sfs_shell` program

We have provided for you a simple shell that will be used to interface with your filesystem and the emulated disk. When grading your work, we will use the shell to test your code, so be sure to test extensively. To use the shell, build the source code using the Makefile provided and run with the `./sfssh` with the name of a disk image, and the number of blocks in that image. For example, to use the `image.5` example given in the `data` folder below, run from the `src` folder after building the source files:

```
./sfssh ../data/image.5 5
```

Or, to start with a fresh new disk image, just give a new filename and number of blocks:

```
$ ./sfssh newdisk 25
```

Once the shell starts, you can use the `help` command to list the available commands:

```
% ./sfssh ../data/image.5 5
sfs> help
Commands are:
format
mount
```

```
debug
create
remove <inode>
cat <inode>
stat <inode>
copyin <file> <inode>
copyout <inode> <file>
help
quit
exit
sfs>
```

Most of the commands correspond closely to the filesystem interface. For example, `format, mount, debug, create` and `remove` call the corresponding functions in the `fs.c`. Make sure that you call these functions in a sensible order. A filesystem must be formatted once before it can be used. Likewise, it must be mounted before being read or written.

The complex commands are `cat,copyin,` and `copyout`. `cat` reads an entire file out of the filesystem and displays it on the console, just like the Unix command of the same name. `copyin` and `copyout` copy a file from the local Unix filesystem into your emulated filesystem. For example, to copy the dictionary file into inode 10 in your filesystem, do the following:

```
sfs> copyin /usr/share/dict/words 10
```

Note that these three commands work by making a large number of calls to `fs_read` and `fs_write` for each file to be copied.

7 Tests

To help you verify the correctness of your SimpleFS implementation, we have provided you, with the following disk images:

```
image.5
image.20
image.200
```

Also note, that depending on how you implement the various functions, the number of disk reads and writes may not match.As long as you are not too far above the numbers in the test case, then you will be given credit.

8 Submitting Your Work

To submit your work, you must use the `cs350_submit` program in the `linux.student.cs` computing environment.

Important! You must use `cs350_submit`, not `submit`, to submit your work for CS350.

Note the usage for `cs350_submit` command is as follows

```
% usage: cs350_submit <assign_dir> <assign_num_type>
```

The `assign_dir` is the path to the root directory that should contain the `C` files for your the final project. If you kept the directory structure in the starter code, this is the `src` folder. You are required to use the `C` files provided in the starter code.

The `assign_num_type` for this final programming assignment is `FINAL`.

Therefore, to run the `cs350_submit` command for submitting the final programming assignment, the command will look like this:

```
% cs350_submit cs350-student/final/src FINAL
```

The argument `assign_dir` in the `cs350_submit` command, packages up your `userspace` program and submits it to the course account using the regular `submit` command.

This assignment only briefly summarizes what `cs350_submit` does. Look carefully at the output from `cs350_submit`.

It is a good idea to run the `cs350_submit` command like this:

```
cs350_submit cs350-student/final/src FINAL | tee submitlog.txt
```

This will run the `cs350_submit` command and also save a copy of all of the output into a file called `submitlog.txt`, which you can inspect if there are problems. This is handy when there is more than a screen full of output.

You may submit multiple times. Each submission completely replaces any previous submissions that you may have made for this assignment.