Instructor: Ashvin Goel
Course Number: ECE344

Home
Lecture Notes
Lab Assignments
Discussion (piazza)
Grades (UofT portal)

Operating Systems

ECE344, FALL 2016
UNIVERSITY OF TORONTO

LAB 6: A SIMPLE FILE SYSTEM

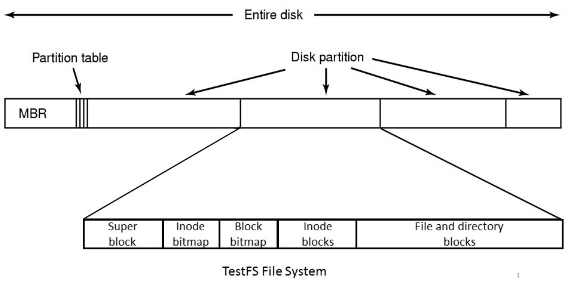
Due Date: Dec 5, 2016, 5 pm

In this lab, you will be developing a file system that supports creating very large files. To simplify the project, we are providing you with the code for a simple user-level file system, called `testfs`, that currently allows creating small files. You will extend `testfs` so that you can create very large files.

FILE SYSTEM BACKGROUND

In this section, we provide an overview of how a file system is organized. Our description will focus on the `testfs` file system, which borrows many ideas from a Unix file system.

The figure below show the layout of a file system on disk. A file system is typically located in a contiguous region on disk called a partition. The disk can have one or more partitions, and each partition has its own file system.



A file system views the disk partition as a array of contiguous blocks. Each block has a fixed size that typically ranges from 512 bytes to 16 KB. The `testfs` file system has a block size of 8192 bytes (8 KB).

The file system uses the block index to locate a block. For example, the first block in the file system is at index 0, the second block is at index 1, etc. We call this index the *block number*.

A file system can be viewed as a tree (or graph) structure that has been laid out in the array of blocks. The root of this tree (not to be confused by the root directory of the file system) is the *super block*. This block defines the type of the file system, stores the size of the file system, and stores various parameters that define the format of the rest of the file system.

The `testfs` file system associates an `inode` structure with each file or directory. This structure maintains various pieces of information about the file, such as the length of the file. We describe this structure in more detail below. The inode structures are laid out in an array in one or more blocks, shown as *inode blocks* in the figure above. They are located using an inode index. For example, the first inode in the inode blocks area has an index of 0, the second inode has an index of 1, etc. We call this index the *inode number*. Since the total number of inode blocks is fixed, the total number of files or directories that can be created is also limited to the number of inodes in the inode blocks area.

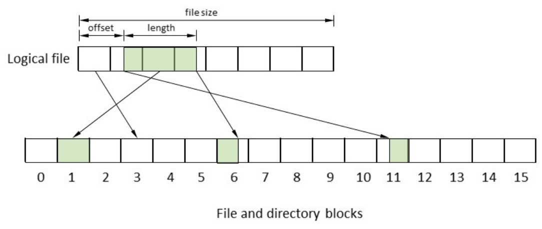
The inode and the block bitmap shown above are also called the allocation bitmaps (the `testfs` code calls them freemaps).

The inode bitmap tracks whether an inode is currently allocated to a file (or directory). When a file is created, the file system uses this bitmap to locate a free inode and allocate it to the file. Bit 0 in the inode bitmap corresponds to inode 0, bit 1 in the inode bitmap corresponds to inode 1, etc.

The figure shows that files and directories are located in the area shown as *file and directory blocks*.

The block bitmap shown above tracks whether the blocks in the file and directory blocks area are currently allocated. When a file needs to grow, the file system uses this bitmap to locate a free block and allocate it to the file. Bit 0 in the block bitmap corresponds to the first block in the file and directory block area, bit 1 in the block bitmap corresponds to the second block, etc. The block bitmap does not track the allocation status of the super block, the allocation bitmaps, and inode blocks because these blocks always remain allocated, i.e., they are created when the file system is created and then they are not freed.

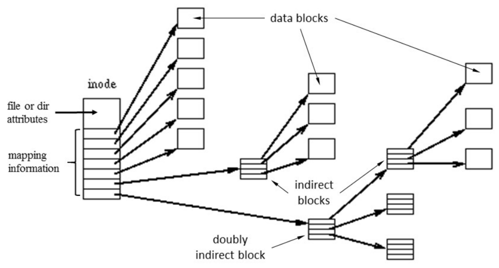
An application or a programmer views a file as a contiguous array of bytes, as shown below. The file size is the number of bytes in the file. Typically, a file is accessed by reading or writing a range of a file. A file range starts at `offset` bytes from the start of the file, and has `length` bytes. For example, the entire file can be read by specifying an offset of 0, and a length equal to the file size.



Each file or directory consists of zero or more blocks, located in the file or directory blocks area. These blocks could be located contiguously but then growing a file would require copying file data (similar to the problems associated with contiguous memory allocation). Instead, the file blocks are located non-contiguously, as shown in the figure above. Thus reading the logical file range requires reading physical blocks 11, 1 and 6.

How should we track which physical blocks are associated with a file? This mapping problem is similar to the paging problem. We need a data structure that maps logical file blocks (shown in the top of the figure above) to the physical blocks on disk (shown in the bottom of the figure above).

The `testfs` file system, similar to Unix file systems, maintains the logical to physical file block mapping using a multi-level tree structure, similar to a multi-level page table. This tree structure is shown in the figure below.



Recall that the `testfs` file system keeps an inode (located in the inode blocks area) for each file or directory. The inode structure for each file forms the root of the mapping for the file. The inode stores various file attributes, such as the size of the file, and then the mapping information.

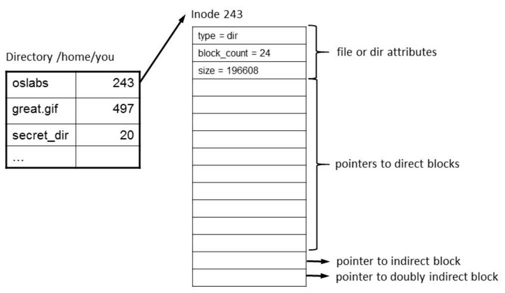
The mapping information consists of three types, direct (or data) blocks, indirect blocks, and doubly indirect blocks. The inode stores block pointers to 10 data blocks (a block pointer is a block number, helping locate the block on disk). These 10 data blocks are associated with the first 10 logical blocks of the file. Since the `testfs` block size is 8192, these block pointers allow reading the file range from byte offset [0, 81920), offset 0 is included, offset 81920 is not. Then the inode stores a block pointer to an indirect block. The indirect block does not store file data. Instead, it stores pointers to data blocks. Each pointer is 4 bytes. Hence, the indirect block stores 8192/4 = 2048 block pointers. This allows reading the logical file block numbers [10, 10 + 2048 = 2058). This corresponds to the file range from byte offset [10 * 8192 = 81920, 2058 * 8192 = 16859136). Finally, the inode stores a block pointer to a double indirect block, which stores 2048 block pointers to indirect blocks. This allows reading the logical file blocks [2058, 2058 + 2048 * 2048 = 4196362). Thus, the maximum file size supported by the `testfs` file system is 4196362 * 8192 = 34376597504 bytes or roughly 32 GB, enough to store most large video files!

Where is all this mapping information stored? The data blocks as well as the indirect and double indirect blocks are stored in the file and directory blocks area. In fact, a block in this area could be allocated as a data block, and later after it is freed, it could be allocated as an indirect block. Similarly, an indirect block can be freed and later allocated as a data block.

Why does the `testfs` file system store the mapping information using this complicated scheme with direct blocks, indirect blocks and double indirect blocks (compare this scheme to a three-level paging scheme, which is similar to only using double indirect blocks)? The reason the file system uses this scheme is that it optimizes accesses to small files. Reading or writing a small file that is less than 10 blocks (80K) requires accessing the inode of the file in the inode block and then the 10 data blocks. No additional indirect block accesses are needed. Once the file becomes larger, then indirect blocks need to be accessed to read the later blocks. Since most files in a typical file system are small (typically 10-20K), this optimization is effective.

Until now we have discussed how a file is stored on disk, but how do we access a specific file. Right now, we would need to know the inode number of the file. This would allow us to access the inode structure associated with the file in the inode blocks area, and then follow the direct, indirect and double indirect pointers to access the data blocks of the file.

Inode numbers are, of course, not easy to remember. We are all familiar with file names, which make it easier to locate files. So we need a mapping from file names to inode numbers. This mapping is stored in directories, as shown in the figure below.



A directory is stored in the file system in exactly the same way as a file. It has an inode and the corresponding block pointers for locating the physical blocks associated with the directory. A block in the file and directory blocks area can be allocated to a file or directory. A file block, after it is freed, can later be allocated to a directory, and similarly, the other way around.

However, unlike a file, the contents of a directory are in a specific format that the file system uses to map file names to inode numbers. Each such mapping is stored in a directory entry (`dirent`) structure, that contains a variable length file (or directory) name, and the inode number associated with the file. For example, the figure shows that the data blocks of the directory `/home/you` (stored in the file and directory blocks area) contain at least 3 directory entry structures. The first such structure has the name `oslabs` and this name is associated with the inode number 243. The inode 243 (located in the inode blocks area) shows that the type of this inode is a directory. This directory occupies 24 physical blocks and has a size of 196608 bytes.

The directory structure enables hierarchical naming. Say we need to get to the data blocks of the `/home/you` directory. The file system keeps the inode of the root `(/)` directory in a well-known location (e.g., inode number 0). The file system reads the data blocks associated with this inode to look up the directory entry associated with the name `home`. This directory entry has the inode number for the `home` directory. The file system reads the data blocks associated with this inode to look up the directory entry associated with the name `you`. This directory entry has the inode number for the `you` directory. This process goes on until the desired file or directory inode is found.

SETUP

Add the source files for this lab, available in `fs.tar`, to your repository, and run `make` in the newly created `fs` directory.

```
cd ~/ece344
tar -xf /cad2/ece344f/src/fs.tar
git status # should say that "fs/" directory is untracked
git add fs
git commit -m "Initial code for Lab 6"
git tag Lab6-start
cd fs
make
```

The make command will create two executables called `mktestfs` and `testfs`.

The `mktestfs` program creates a new `testfs` file system. It takes one parameter, the device, partition or a Linux file, on which to create the file system. Since we do not have direct access to a physical device or disk partition, we will create a `testfs` file system inside a single Linux file. **Make sure to provide a new file name, or else `mktestfs` will overwrite your file.**

The `mktestfs` program also takes one optional parameter, which specifies the maximum size of the file system, in MB.

```
cd ~/ece344/fs
./mktestfs device
```

The following command will create a Linux file called `device` that serves as the disk partition for the `testfs` file system. How does using a Linux file as a disk partition work? The `testfs` file system reads and writes this "disk partition" using "physical block numbers". The Linux file system translates these block numbers to the actual physical block numbers on disk where the file blocks are located. Hence, two translations are performed to access each `testfs` file block, one by the `testfs` file system, and the other by the Linux file system.

Go ahead and browse the `mktestfs` program. You will see that it creates the super block, the allocation (inode and block) bitmaps, and the inode blocks on disk. It also initializes the root `/` directory on disk. Then it prints some file system statistics.

The `testfs` executable is the file system program. You run this program with the device created by the `mktestfs` program.

```
cd ~/ece344/fs
./testfs device
```

This command will take you to an interactive shell. Type "help" to see the various, simple commands supported by the `testfs` file system. Try these commands to see how they work.

Note that to list the files in the current directory, you will need to type `ls .`, where the dot following the `ls` command refers to the current directory.

MULTI-BLOCK READ-WRITE SUPPORT

The basic `testfs` file system that we have provided has a severe limitation. While the file read and write interface work at the byte granularity (read a file range from any byte offset, up to any byte length), the current `testfs` implementation does not support reading across block boundaries. Hence, it will allow reading a file range say [8000, 8190) but not [8000, 8200) because the block size is 8192 and the latter range crosses a block boundary.

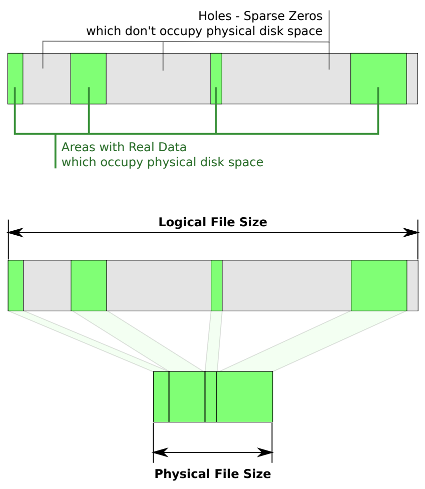
In the first part of the lab, you will fix this problem, so that arbitrary file ranges can be read or written.

LARGE FILE SUPPORT

The `testfs` file system is designed to support large files (32 GB) as discussed earlier. Unfortunately, currently it only supports indirect blocks and not double indirect blocks. As a result, the maximum file size is limited to 16859136 bytes or 16 MB, as discussed earlier. That is a pretty small file size these days. Your next (and final) job for this lab will be to implement the double indirect block functionality to support large files.

You will need to implement reading and writing of data blocks that are located via the double indirect block. You will also need to implement freeing of these data blocks when a file is removed.

You might be wondering how you will get the disk space on the UG machines to test your large files. Fortunately, the `testfs` file system supports *sparse* files. These are files that can be large without necessarily storing all the data on disk. How is that possible? In a sparse file, when a block is empty (it has never been written), the block is not stored on disk, as shown below (figure from wikipedia).



To track an empty block, the block pointer in the inode associated with this block has a special value (e.g., 0). The block is allocated when it is written for the first time.

When an empty block in a sparse file is read, the file system transparently returns a block filled with zero bytes (the ascii NULL character, not the ascii '0' character). The application does not know that the file system does not store this data on disk.

With a sparse file, an application can create very large files by writing, say a single byte, close to the end of the maximum file size. In this case, the file is considered very large but it would only store one data block on disk (however, it will need to store a double indirect and a indirect block to locate this data block).

ERRORS, ERROR, ERRORS

File systems should be made robust to various types of errors. Unlike an error in memory, which can be fixed by restarting a program or rebooting the machine, an error caused by the file system software will propagate to disk and stay there forever, perhaps crashing programs, or causing security vulnerabilities.

Your job is to ensure that your modified `testfs` can handle two important kinds of errors correctly.

The first error is a write to a file beyond the maximum file size. In this case, your code should return the `EFBIG` error to indicate that the file read or write is to an offset that is bigger than the maximum file size. Note that if a write succeeds partially, for example, some of the blocks are written, then the file size may need to be increased, even though the error is returned.

The second error is a more insidious error that occurs when there is no more space available in the file system. Unless the file system is designed carefully, this error can cause data corruption or deadlocks. Fortunately, `testfs` is single threaded and so deadlocks are not possible. In our case, your code needs to ensure that reads and writes carefully return the `ENOSPC` error. Look at where this error is generated in the code. Your code needs to simply propagate this error correctly. Again, if a write succeeds partially, then the file size may need to be increased, even though the error is returned.

Note that the convention in the code is that we return a negative value when an error occurs. So, for example, we return -EFBIG for the maximum file size error.

HINTS AND ADVICE

We recommend understanding how the code that we gave you works. We provide the following files:

- **testfs.c:** Contains the main() function for the `testfs` code. This code parses the interactive commands and invokes the appropriate `testfs` commands.
- **super.c:** This file is used to perform file system wide functions such as creating the super block, allocating blocks and manipulates the allocation bitmaps.
- **block.c:** Read and write physical blocks to the disk device.
- **bitmap.c:** Bitmap manipulation routines.

- **dir.c:** All the directory related functionality is implemented in this file.
- **inode.c:** Create and remove inodes. This file also has functions for caching the disk inode structure in memory so that if a file is accessed frequently, its inode can be accessed efficiently.
- **read_write.c:** Read and write data from a file. When a file is removed, free the blocks of a file. **This is the only file that you should modify for this lab.**

TESTING YOUR CODE

When testing your code, running `testfs` interactively eventually becomes cumbersome because you need to type all the commands again and again.

Running testfs non-interactively

However, `testfs` can be run non-interactively as follows:

```
  echo "ls ." | ./testfs -n device
```

The `-n` option disables printing of the `testfs` interactive prompt. The `testfs` program also takes a `-v` option that prints out each command that is being run.

If you want to run multiple commands non-interactively, the easiest option is to create a shell file. Here is an example:

```
#!/bin/sh

echo "ls .
create file1
stat file1" | ./testfs -n device
```

Save the text above in a file, say `mytest`. Notice the double quotes start on the first line, and end on the last line. Then make this shell file executable as follows.

```
chmod +x mytest
```

Now you can run `testfs` non-interactively by simply typing:

```
./mytest
```

On a newly created `device`, here is the output of the script:

```
./
../
file1: i_nr = 1, i_type = 1, i_size = 0, block_count = 0
```

You can look at the scripts described below for writing more powerful shell scripts.

Testing scripts

To help you test your code better, we have provided several scripts, as described below.

Please run these tests from the `fs` directory.

All tests send their output to a file. They print a PASS when this output file matches the expected output file, or else they print FAIL.

For example, the `./tests/test_rw` test described below sends it output to the `test_rw.out` file and its expected output is in the `./tests/test_rw.txt` file.

These tests will create a `testfs` file system image on a file called `device`. **If you had previously created this file, it will be overwritten.**

./tests/test_rw:
: This reads and writes to direct and indirect blocks, at multi-block granularity. It will pass once multi-block read/write is implemented.

./tests/test_rw_large:
: This tests reads and writes via the double indirect block. It also requires implementing the removal of large files correctly.

./tests/test_rw_too_big:
: This tests writes beyond the maximum file size.

./tests/test_rw_no_space:
: This tests for writes when the file system is full.

You can test your entire code by using our auto-tester program at any time by following the testing instructions. For this assignment, the auto-tester program simply runs the scripts described above.

Checking the consistency of the `testfs` file system

We have provided a program called `cktestfs` that checks the consistency of the `testfs` file system image. You can run this program as follows:

```
cktestfs device
```

Make sure to run this program only after you have exited from the `testfs` program.

This program checks that the file system data structure on disk has no obvious inconsistencies. For example, it checks that the blocks that are considered allocated (as per the block bitmap) belong to some file or directory (i.e., some file or directory has a pointer to this block), or else the block has been leaked (it can never be deallocated).

Furthermore, a block should only be pointed to by one pointer in a file or directory. Otherwise, two files would refer to the same block, and any modification to one file would affect another file.

Similarly, `cktestfs` checks the consistency of the inode bitmap by checking that an allocated inode is referred to by some directory entry, or else the inode has been leaked.

After you make your modifications, it will be good idea to check that your file system is consistent by running this program.

The tests above run the `cktestfs` program for you.

USING GIT

You should only modify the following files in this lab.

```
  read_write.c
```

You can find the files you have modified by running the `git status` command.

You can commit your modified files to your local repository as follows:

```
  git add read_write.c
  git commit -m "Committing changes for Lab 6"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time, if needed.

Once you have tested your code, **and committed it** (check that by running `git status`), you can tag the assignment as done.

```
git tag Lab6-end
```

This tag names the last commit, and you can see that using the `git log` or the `git show` commands.

If you want to see all the changes you have made in this lab, you can run the following `git diff` command.

```
git diff Lab6-start Lab6-end
```

More information for using the various git commands is available in the Lab 1 instructions.

CODE SUBMISSION

Make sure to add the `Lab6-end` tag to your local repository as described above. Then run the following command to update your remote repository:

```
git push
git push --tags
```

For more details regarding code submission, please follow the lab submission instructions.

Please also make sure to test whether your submission succeeded by simulating our automated marker.