

A Simple File System DLX/OS

Prof. Darrell Long[†]
Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Design Due: November 19, 23:59
Code Due: November 26, 23:59

1 Ground Rules

The basic rules and project information are the same as those for the last assignment. You may work on this assignment either by yourself or with a single project partner (*i.e.* a group of two people doing a single project). If two people partner on a project, include both names and CATS accounts in any files either of you modified. A project team should only turn in one copy of their assignment: the person who is not turning in the assignment should submit a single file called `partner` containing the name and CATS account of both project members into that directory. The grade received by each project partner will be the same.

You may now discuss your design with other members of the class. But, if you do so, you must follow the Gilligan's Island rule – you may bring no notes of any kind to, or from, the discussion and you must wait at least $\frac{1}{2}$ hour before sitting down to work on your code. Also, you must note in your design document any discussions that you had, with whom you had them, and what you discussed.

Remember: you cannot share code, view another's code, or allow others to view your code. Failure to comply will be considered a violation of the ethics portion of this class, with all that implies.

As with other projects, the following suggestions apply:

- Start early,
- Read and understand the relevant DLX/OS code before doing anything, and
- Do your design document before writing a single line of code.

2 Basics

The goal of this project is to implement a simple file system in DLX/OS. The file system should have the following features:

1. The file system has to support the following calls. The actual location of the implementation is described in the specifications section.
 - `fileID = Open(filename, mode)`
 - `retval = Close(fileID)`
 - `bytesread = Read(fileID, buffer, bytesToRead)`
 - `byteswritten = Write(fileID, buffer, bytesToWrite)`

[†]Parts of the assignment have been taken from those of Ethan Miller and Scott Brandt.

- `retval = Seek(fileID, offset, whence)`
 - `retval = Delete(filename)`
2. There is a single directory in which files reside.
 3. The files should be allocated using a File Access Table (FAT), like that of MS/DOS, and described on pp. 438-442 in Tanenbaum's *Modern Operating Systems*. You may use either FAT-12 or FAT-16, although the later is probably much easier.
 4. User traps corresponding to the above calls should be implemented. There are stubs for these calls in the latest version of `traps.c`: you may need to ensure that the corresponding calls are in `usertraps.s`. Of course, you also have to make sure these calls actually call `FsOpen()`, `FsRead()`, `FsWrite()`, ...
 5. You should write a DLX program that lists all of the files in the (single) directory. This can be done easily by treating "/" as a file, allowing it to be opened and read. Then, your user program need only be able to parse the contents.

2.1 Design Document

The assignments in this class do not require you to write large quantities of code. For most assignments, you need only write a few hundred lines of code, usually fewer. However, the concepts covered in class and in the operating system are quite difficult for most people. As a result, deciding which lines of code to write is very difficult. This means that a good design is crucial to getting your code to work, and well-written documentation is necessary to help you and your group to understand what your code is doing.

The most important thing to do is write your design first and do it early! Each assignment is about two weeks long; your design must be complete by the end of the first week. Doing the design first has many advantages:

- You understand the problem and the solution before writing code.
- You can discover issues with your design before wasting time debugging code that you'll never use.
- You can get help with the concepts without getting bogged down in complex code.

Doing your design early is the single most important factor for success in completing your programming projects!

3 Details

The operating systems hasn't changed, so you can continue to use the code you've been working on. But if you've had trouble getting things to work you may want to start fresh with a clean copy of the operating systems.

Your file system calls will be integrated into DLX/OS in `filesys.c`. There are routines called `FsDlxRead()`, and so on for the other calls you need to implement. You need to modify these calls to use the file system you are designing. Your routines will automatically be called when you call `FsOpen()` with a file name that starts with "dlx:". For example, calling `FsOpen()` on the file "dlx:/foo" will result in `FsDlxOpen()` being called with the file `/foo`. Other calls (except for `FsDelete()`) are switched automatically based on the file identifier set in `FsOpen()`, which remembers which file system (UNIX or DLX/OS) the file was opened in.

Reading and Writing Disk Blocks

For this assignment, you'll be reading and writing blocks from a file called (by default) `dlxdisk` (you should allow its name to be supplied by the `-B` argument passed to the operating system) using the normal file system calls from within DLX/OS (`open()`, `read()`, `write()`, ...). You should treat the `dlxdisk` file as a "raw disk" on which to build your file system. Your operating system need not initialize the disk: you may write a regular program (running in UNIX) to set up any initial data structures on `dlxdisk` that your OS might need. Your disk should be persistent – if you shut down DLX/OS and restart it, DLX/OS should still be able to use the file system with all of the changes that the previous run made to the file system (creates, writes, deletes, ...).

File blocks should be 1KB (1024 bytes) long. This means that a `read()` call to read bytes from the DLX/OS disk looks like this:

```
#define DLX_DISK_BLOCKSIZE 1024
unsigned char buf[DLX_DISK_BLOCKSIZE];

nbytes = read(dlxdiskfd, buf, DLX_DISK_BLOCKSIZE);
```

Write calls look about the same.

File Allocation Table (FAT)

You should implement a File Allocation Table similar to that of MS/DOS (described on pp. 438-442 of the Tanenbaum text). A FAT is essentially a set of linked lists embedded in an array. One of these linked lists is the list of free blocks. One of these linked lists represents the file that is the root directory. Both of those linked lists are special and must be easily found in a known location: using a *super block* at the beginning of the disk is a good way to accomplish this goal.

Consistency is the most important issue in file systems. A file system can be arbitrarily fast, but if it loses data then it is useless. When allocation and deallocating blocks, you should always err on the side of safety, preferring to lose blocks (a block is free but not included in a file) rather than allowing a block to be allocated twice. The order in which you do things is particularly important, so at each step of your algorithms, try to think of what would happen if the power were suddenly interrupted – a free block marked as allocated (not too bad), or an allocated block marked as free (very bad).

Directory

You must support a single directory that can accommodate file names using the standard MS/DOS “eight plus three” format *if you are working alone* while if you are working with a partner, you must implement directories with file names of up to 80 characters. For long file names, you may not waste space – a 4 character file name shouldn’t occupy 80 characters of space in the directory. For each file in the directory, you should mimic the MS/DOS directory entry format, except that you only need to maintain: name, date & time, pointer to the start of the file, and a length of the file (in bytes). Of course, the directory itself is like a file, and uses the FAT to store pointers to all of its blocks.

The root directory, as in UNIX, is referred to as “/”. This means that the call to open the file `foo` in DLX/OS is `FsOpen("dlx:/foo", ...)`. Similar naming applies to the delete call.

- **(Individual)** Implement simple MS/DOS “eight plus three” file names.
- **(Partners/Extra Credit)** Implement long file names, up to 80 characters in length.

Super Block

The *super block* contains information about the file system (metadata). For example, it will contain a pointer to the FAT, a pointer to the free list embedded in the FAT, and a pointer to the root directory.

Opening Files

There are three possible flags to open: **read**, **write**, and **create**. The **read** and **write** flags are used to indicate which operations are permitted on an open file. The **create** flag is set by the caller when the user wants to create a file if it’s not already found. If **create** isn’t set and the file isn’t found, an error should be returned. If **create** is set and the file doesn’t yet exist, it should be created with 0 size. Much of the open code is already written for you.

Reading and Writing Files

Reading files is self-explanatory. Your file system simply needs to find the appropriate data and read it in. Writing is a bit more complex – you may need to allocate a new block. Otherwise, data is written as you’d expect.

File reads and writes need not be on file block boundaries.

Seeking

Seeks in the DLX/OS file system work just the way seeks work in UNIX, with the exception of seeking past the end of file. A seek updates the “current” pointer but doesn’t transfer data. Seeks past of the end of the file fail and return a (negative) error code.

Error Conditions

Your file system should catch errors and return appropriate error codes. Possible errors you may want to catch include:

- File not found (*open, delete*),
- Out of space (*open/create, write*),
- File not open (*read/write*) – this may already be caught,
- End of file (*read*), and
- Seek past end of file (*read, write*).

(Extra Credit) Directory Listing

You need to write a DLX *user* program to list all of the files in the (single) directory. It should mimic the `ls` command of UNIX in its simple cases. Keep in mind that listing file sizes will likely require an additional call similar to the UNIX `stat()` system call.

3.1 Hints

- Implement the pieces in the following order:
 1. UNIX program to build a file system on `dlxdisk`. Use this to make sure your file system design makes sense. You will also want to build a UNIX program to list the contents of the file system and other utilities (trust me, you *will* want this utility).
 2. Manage the FAT on disk. You may want to cache a copy of this in memory. Remember that the FAT is simply a set of linked lists embedded in an array.
 3. Read existing files. You may use your UNIX utility to create files to test this feature.
 4. Create and write files.
 5. Delete files.
 6. Directory listing utility.
- Use a UNIX program to debug the contents of your disk. You may not be able to follow what’s happening in your operating system that easily, but you can certainly analyze what the operating system does to the disk.
- Try your file system code with both small (≈ 1 MB) and large (≈ 20 MB, or as large as your paltry CATS quota will allow) file systems. Your UNIX program should be able to build a `dlxdisk` of any size (given the limitations of FAT); the only change should be the length of the FAT and the file system size at the start of the disk (a *super block*).
- Your root directory must be easily located, perhaps via a pointer in the super block.
- Make sure you close the `dlxdisk` file before exiting DLX/OS! Failure to do so may cause some data to be lost.
- Almost all of your code will be in `filesys.c` and `filesys.h`, with a bit of code in `traps.c` and `traps.h`.
- Use debugging statements or instruction and data traces (see the `dlxsim` page) to track where your code is going.

4 Deliverables

A compressed `tar` file of your project directory, including your design document. You must do “`make clean`” before creating the `tar` file. In addition, include a `README` file to explain anything unusual to the TA or grader (for example, testing procedures). Your code and other associated files must be in a single directory so they will build properly in the submit directory.

As with other projects, you’ll need to hand in your design documentation and your code. Because your code may include both operating system code, user programs, and `UNIX` programs, please make sure your `Makefile` builds them separately. For this project, you’ll need to be particularly careful because you might have programs that need to be compiled with regular (not `DLX`) `gcc`. *Do not hand in `dlxdisk`!*

Remember: Do not submit object files, assembler files, or executables. Every file in the submit directory that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade.