ECE 595 Fall 2018 Lab #4: File Systems

Due Sunday, December 2, 2018 at 23:59 EST

Get the examples to ostest from ~ece595/labs_2018/Labs/lab4-ostests-example.c

Introduction

The purpose of this lab is to become familiar with file systems and their implementation. In particular, you will accomplish the following in implementing a UNIX-like file system called DlxFS (DFS for short):

- implement a DFS system driver,
- implement fdisk to format a disk with the DFS filesystem,
- implement common libc-style file I/O functions,
- and implement a filesystem buffer-cache.

The key to successfully finishing this lab is to follow all the keys that have been given in previous labs: checking return values, well-thought-out testing plans, clear debugging statements, and lack of procrastination.

Background and Review

File Systems in General

A file system is simply a means of organizing a large set of bytes using a small amount of overhead, both in terms of size and access speed. There is no restriction as to what type of hardware a file system runs on: it could be Read-Only memory like a DVD-ROM, it could be a traditional magnetic or solid-state hard drive, it could be a RAM-based file system that is entirely in volatile memory, or it could even be a virtual disk that resides in a file (as what you will be implementing in this lab). In this way, the file system can shield the low-level details of hardware disk manipulation from higher-level programs.

File System Terms: Paging Revisited

A file system is similar in nature to memory paging. There is a concept of a *file*, (similar to the concept of a process's *virtual address space* in paging) which is a set of contiguous bytes. However, as you learned with paging and segmentation, it is difficult to store sets of contiguous bytes directly. Therefore, the filesystem breaks both the files and the disk up into *blocks* (analogous to virtual pages and physical pages), and keeps a table (similar to a page table) that translates the virtual block number of a file to a block on disk (or a block in the file system, this subtlety in terms will become clear soon).

Recall that with paging, you needed to store the page table and other information about a process's virtual address and physical page usage. You stored this information in a *page table*, and the pointer to the page table is stored *process control block* (PCB) structure. For file systems, we need a similar type of structure to store the information about a file: things such as the filename, the block translation table, length of the file, etc. This structure is called an *index node* or *inode* for short. An inode is the metadata associated with a file, meaning that each file has one inode, and each inode represents one file. Note that, inode structure in this lab is different than inode in unix. For example, unlike unix, we are storing the fileName in the inode in this lab.

In lab 3, you kept track of which pages were free and which were in use through the use of a bitmap where each bit corresponded to 1 page. This was called the *freemap*. We will use the same concept in this lab to keep track of which file system blocks are in use and which are free, except that we will call it the *free block vector* in this lab instead of the freemap.

The table below summarizes the relationship between paging terms and file system terms:

File System Term	Paging Equivalent
block	page
file	process virtual address space
inode	page table/process control block
free block vector	freemap
block number	page number
physical block	physical page
virtual block	virtual page
file system block	<no equivalent>

One major difference between paging and file systems is that with paging, we had the luxury that when the system turned off, we had no need to save any information about the state of memory. It was assumed that when the machine turned on, it would rebuild all the paging infrastructure from scratch again. With file systems, this is not the case. Therefore, we must write all of this file system information to the disk before exit, and read back in when the operating system starts.

One of the primary issues when dealing with a physical disk is access time. It can take many thousands of times longer to read a byte from a physical disk than it does to read a byte from memory. For this reason, file systems generally employ some level of caching of disk data in memory. For instance, since inodes may be accessed frequently, it makes sense to keep them in memory to allow for fast access.

As with paging, there are several important terms involved with file systems whose definitions must be clear:

- **physical block**: a consecutive chunk of bytes on a physical disk. It is identified by the physical block number. This is the native number of bytes that can be read or written at a time to the physical disk. Note that in practice this is often called a *logical disk block*, as modern disks (e.g. SCSI) can mask off bad blocks, and present to an operating system the abstraction of a one-dimensional array of consecutively numbered logical blocks.)
- **file system block**: this differs from a physical block in that a filesystem can be formatted to use blocks that are a different size than the physical disk. Its read/write functions must account for the difference in size. For instance, if a file system is formatted to use 1024-byte blocks and the disk uses 512-byte blocks, then block number 5 in the filesystem will actually correspond to blocks 10 and 11 on the physical disk.
- **virtual block**: a virtual block is used by inode-based functions. Similar to virtual addresses in paging, a virtual block number is translated to a file system block number using the translation table stored in the inode.
- **inode**: a file-system-specific structure that stores information about files.
- **free block vector**: a bitmap representing which file system blocks are in use.
- **superblock**: a special structure that stores information about the formatting of a given file system on a disk. This includes things such as where the inodes are stored, where the free block vector is stored, how many inodes there are, etc.

One important point to note is that the physical disk is only able to read or write exactly one physical block at a time. This means that if a program wants to overwrite 4 bytes in a file, and the physical block size is 512 bytes, the file system it first has to read the existing 512 bytes from the disk, change the 4 bytes in the copy of the block in memory, then write the entire 512 bytes back to the disk.

Disk functions vs. DFS functions vs. DfsInode functions vs. File functions

It is easy in this lab to get confused between disk operations, file system operations, and file operations. Disk operations are performed by the actual physical disk hardware. These operations include reading and writing blocks, and reporting disk information such as the total size and the block size.

File system operations (a.k.a. DFS operations) are performed by the file system driver in the operating system. DFS operations include opening the file system (i.e. loading it into memory), closing the file system (i.e. writing it back to the disk), allocating and freeing DFS blocks, reading and writing DFS blocks, reading and writing virtual blocks (through inodes), and block caching.

File operations are performed by a special file library outside the file system driver. File operations include fopen, fclose, fread, fwrite, and fseek. These functions maintain a concept of a "current position" in a file, as well as provide an abstraction for programs that allows them to perform file I/O without having to know anything about inodes.

DLXOS File System (DFS)

The DLXOS File System uses a configurable block size, with the restriction that the file system block size must be an integer multiple of the physical block size. This keeps all file system blocks aligned with physical blocks. The file system layout is configurable based on the total physical disk size and the maximum number of inodes. An example structure for a 64MB physical disk, inode size 128 bytes, and a maximum number of 128 inodes, is as follows (all block numbers are file system block numbers):

- Block 0: master boot record and superblock. Refer to the paragraph below for details.
- Blocks 1 to 16, inclusive: array of inode structures.
- Blocks 17 to 24, inclusive: free block vector
- Blocks 25 to 65535, inclusive: data blocks

File system block 0 is actually represented as two sections: the first section is the "master boot record", where a normal file system would store information necessary to boot an operating system, and the second section is the superblock structure. The location of the master boot record is an agreement between the machine hardware/firmware (e.g. BIOS) and the operating system. Therefore, it must always reside in physical block 0 (note *physical block 0*, not file system block 0, since the hardware knows nothing about our file system). Since we won't be booting from this disk, we will write all zeros to this block.

Also, since the file system driver must read the file system blocksize from the superblock (recall this is configurable), then it must also know exactly which physical block contains the superblock. After reading the superblock, it can then operate in terms of file system blocks. Therefore, the superblock will always reside at physical block 1, right behind the master boot record. The inodes start in the first file system block (NOT physical block) after the superblock, and the free block vector starts in the first file system block after the inode blocks.

To make debugging simpler, a DFS inode structure must be exactly 128 bytes large. Since the filename for an inode is stored in the structure, you can adjust the maximum filename length such that the overall size of the inode is exactly 128 bytes.

The following items should be stored in the superblock (you may add more if you like) (you may declare in include/dfs_shared.h):

- a valid indicator for the file system
- the file system block size
- the total number of file system blocks
- the starting file system block number for the array of inodes
- the number of inodes in the inodes array
- the starting file system block number for the free block vector.

The following items should be stored in the inode structure (you may add more if you like, just remember that the total size must be 128 bytes) (you may declare in include/dfs_shared.h):

- an in use indicator to tell if an inode is free or in use
- the size of the file this inode represents (i.e. the maximum byte that has been written to this file)
- the filename, which is just a string
- a table of direct address translations for the first 10 virtual blocks
- a block number of a file system block on the disk which holds a table of indirect address translations for the virtual blocks beyond the first 10.
- a block number of a file system block on the disk which holds a table of double-indirect address translations for the virtual blocks beyond the first 10 and blocks under single-indirect table.

The direct address translation table in the inode contains file system block numbers for allocated file system blocks that belong to this file. Since most files in any file system tend to be very small, keeping the first 10 translations in this table makes the address translation faster for most files.

In order to keep the size of the inode small, however, large files are going to need additional space beyond the first 10 virtual blocks. Therefore, each file that grows beyond the first 10 virtual blocks will allocate an entire file system disk block to store the rest of its translation table. We will also use double-indirection to store the files that grow beyond max. size supported by single-indirection. In DFS, a file is not allowed to grow larger than is supported by the maximum number of entries in both the direct, indirect and double-indirect tables.

The following items should be stored in the file descriptor datastructure (feel free to edit/delete) (you may declare in include/files_shared.h):

- an in use indicator to tell if the descriptor is free or in use
- the filename, which is just a string
- inode, which this file-descriptor corresponds to
- eof: Indicator if the End-of-file is reached; useful for read operations
- mode: set while opening the file. Possible values: "r", "w"
- current position: Current position in the file
- Process id: Process that opened the file. No other process can do any operations

The DFS driver is in charge of which information is loaded into memory and which is read from the disk on demand. For simplicity, we do not deal with filesystem reliability in the presence of crash failure in this lab. Specifically, to support fast access, when the operating system starts, the DFS driver should load the superblock, all inodes, and the free block vector into structures in memory. It should then mark the superblock on the disk as invalid. When the operating system exits, it should write all the inodes and the free block vector back to the disk, and then write the superblock back to the disk marked as valid. If the system crashes before writing the file system information back to the disk, then the file system is considered corrupted.

Buffer Cache

Research has shown that in practice, disk accesses exhibit *temporal locality* in the sense that if you read/write from/to a block at some point in time, you are likely to read/write from/to that same block again in the near future. This observation indicates that caching recently accessed disk blocks in memory could significantly speed up the average filesystem access time.

A buffer cache is just an array of file system blocks cached in the physical memory. When a read request for a block comes in, the file system first checks the list of blocks already in memory to see if this block has already been read. If it has, it just reads the copy from memory. If it has not, it looks for an available slot in the list of buffer slots. If there is no available slot, it chooses a block to evict from the buffer, and reads the newly requested block into that slot.

The eviction policy could be one of many choices, however one of the most common eviction policy is called "Least Frequently Used" (LFU). When a buffer slot is needed, the block which has been used the least number of times will be taken out.

When a write request for a block comes in, there is also a choice as to what to do. The write operation can write to a buffered block in memory and simply mark that block as "dirty" to be written to the disk when it is evicted from the buffer (write-back), or it can write to the buffered block in memory and write it to disk immediately, or it can write to the buffer and mark the block "dirty" to be written back to disk by a background process that periodically writes dirty blocks to the disk. Most modern file systems use the periodic writing background process as it is an acceptable tradeoff between speed and reliability.

For simplify, we will be implementing a write-back cache, which only writes a dirty block back to the disk when it is evicted, or when the file system is closed. Note that when evicting a block from the buffer, it is only necessary to write it to the disk if it is marked as dirty.

Changes to DLXOS Source

Physical Disk Simulator

We have written a physical disk "simulator" for you in os/disk.c and include/os/disk.h. This simulator uses built-in functions in filesys.c that read and write to an actual file on the Linux system as a disk image. It supports the following functions:

- int DiskBytesPerBlock(): this function returns the number of bytes in one physical block.
- int DiskSize(): this function returns the number of bytes in the entire disk.
- int DiskCreate(): this function creates the Linux file that will hold the file system.
- int DiskWriteBlock(uint32 blocknum, disk_block *b): this function writes one physical block of data to the file at the specified block number.
- int DiskReadBlock (uint32 blocknum, disk_block *b): this function reads one physical block of data from the file at the specified block number.

IMPORTANT: the name of this Linux file is #define-d in include/os/disk.h. You must change this name before the simulator will work.

IMPORTANT: change the DISK_NUMBLOCKS in include/os/disk.h. You should set it to a value that makes disk size to 64MB.

Your file system file must reside in the /tmp directory, and must be named ece595XX.img, where "XX" is your group number. The reason this is required is that the /tmp directory resides on the local hard disk of the computer you are logged in to. Your home directory resides on an NFS file share on a remote server. If all your groups use a file in your home directory as your file system, the NFS server has been known to be slowed down to a halt (e.g. as if under a DDoS attack) as you all keep writing and reading the large "disk" files over and over, especially the night before the assignment is due. Therefore, you have to store the file in a local location. Since you are storing it in a local location, if you all use the same file name, then if two groups are logged in to the same physical machine, the two groups will keep overwriting each other's file (or keep getting access permission denied). Therefore, you have to change the filename to be in /tmp and to be unique to your group.

Blockprint Script

Your primary means of debugging in this lab will be to look at the bytes in your filesystem file to see if they are correct. To do this, you must have the ability to easily look at a binary file. To that end, we have written a simple bash script that uses the xxd utility and sed to allow you to look at specific "blocks" of a binary file, specified by block number. This script resides in the scripts/ directory. You do not have to use this script: it is only provided for your convenience. You can use the script in one of three ways:

```
$ blockprint /tmp/ece59501.img
```
This will print all the blocks of the file /tmp/ece59501.img to the screen.
```
$ blockprint 1 /tmp/ece59501.img
```
This will print physical block 1 of /tmp/ece59501.img to the screen (the superblock).
```
$ blockprint 34 37 /tmp/ece59501.img
```
This will print physical blocks 34 to 37, inclusive, to the screen (the free block vector).

Note that the block size is configurable by changing a variable in the script if you prefer to print DFS-sized blocks instead of physical-sized blocks.

Operating System Tests

We added a convenient user trap, application, and os source file for enabling a user program to run a set of tests within the operating system. Since there is a lot of work to be done in this lab before you can use a user program to read/write files, we require you to test some intermediate functions (such as inode-based functions), which are not visible to user programs. Therefore, the apps/ostests user application is a one-line program that calls the run_os_tests() trap. This triggers the function in os/ostests.c called "RunOSTests()". You can put any sequence of internal OS testing code here that you like, then run the ostests user program to see it run.

GracefulExit replaces exitsim()

We have replaced all the calls to "exitsim" in the simulator code with calls to "GracefulExit", which is a new function defined in traps.c. This function simply tries to first close the filesystem, then calls exitsim. This functionality is required since the disk copy of the filesystem is invalidated until the file system is closed.

New OS Files for Your Code

You will need to fill in os/dfs.c, os/files.c, include/dfs_shared.h, include/files_shared.h, include/os/dfs.h, and include/os/files.h with your code for this lab. The dfs-named files deal with the DFS file system driver, and the files-named files deal with the file I/O interface to user programs (think fopen, fclose, etc.).

Shared OS and user header files

You will be writing two user programs in this lab: one to format a disk and one to test file reading and writing. These programs and the operating system will need to share some information such as the inode and superblock structures, file descriptor structures, etc. To facilitate this, user programs #include the shared versions of the header files (in include/), and os programs #include the os-specific versions of the header files (in include/os/), and these os-specific header files #include the shared header files at the top. You should not share any information with user programs that is not necessary, such as os-level function prototypes, internal OS #define-d constants, etc.

#define-d Constants vs. Superblock Information

It is very tempting to #define all the relevant information about the file system. For instance, the user program that formats the file system can #define the block size. It would be tempting to #define this in the include/dfs_shared.h header file, and use the same constant in the operating system. However, since the block size is configurable, if I use your file system driver on a filesystem that I created with a different block size, it should still work. If you use your #define-d constant, you will be using the wrong block size. Recall that the block size (and all other configurable file system options) are written in the superblock structure. Your driver therefore should use the values in the superblock structure after it is read from the disk, rather than a constant.

You may ask, however, then how are you supposed to know how large to make your dfs_block stuctures and inode and free block vector arrays at compile time, if you won't know how big any of them are until you read the superblock at run time? The answer is that you must assign a #define-d maximum value to each of these items, and then use however many items/bytes that the superblock requires at runtime.

When grading, we will take off points for using any #define-d constants where the superblock information could be used instead.

Assignment

Download and untar ~ece595/labs_2018/Labs/lab4.tar.gz. This will create a `lab4/` directory for you. Put all your work in this directory structure. **DON'T FORGET TO CHANGE THE GROUP FILENAME FOR YOUR FILESYSTEM.**

1. (10 points) **Implement a user program named "fdisk" to format your disk**. You can use some existing user traps for the disk simulator that will allow you to write blocks, read the disk size, and read the disk blocksize. Your file system should be formatted as outlined above in this lab handout. Use the following options:
    - your inode structure should be 128 bytes
    - you should have 128 inodes in your filesystem
    - you should have 10 direct-addressed entries in the inode's virtual block translation table
    - you should have one indirect-addressed block that is only allocated as needed
    - you should have one double-indirect-addressed block that is only allocated as needed

    You can use disk_write_block (int phyblock, char* data) in FdiskWriteBlock function to write to the disk (not dfs).

2. (20 points) **Implement a DFS file system driver in DLXOS**. Write the following functions in dfs.c. Note that the inode-based functions will be implemented later. Be sure that you minimize the amount of information that is shared between the os and the user programs through dfs_shared.h if that information could be better shared through the superblock.
    - void DfsModuleInit(): initialize and open the filesystem
    - int DfsOpenFileSystem(): read the superblock, inodes, and free block vector from the disk, then invalidate the disk's copy. Return DFS_FAIL on failure, and DFS_SUCCESS on success. This fails if the filesystem is already open.
    - void DfsInvalidate(): invalidate the current memory copy of the filesystem. You'll need this in order to be able to format a disk.
    - int DfsCloseFileSystem(): write the superblock, inodes, and free block vector back to the disk. Write the valid superblock last to make sure the other actions succeed. Return DFS_FAIL on failure, and DFS_SUCCESS on success. This fails if the filesystem is not open.

- int DfsAllocateBlock(): Allocate a file system block, returning its block number on success, DFS_FAIL on failure. This fails if the filesystem is not open.
- int DfsFreeBlock(int blocknum): Free a file system block. Return DFS_FAIL on failure, and DFS_SUCCESS on success. This fails if the filesystem is not open.
- int DfsReadBlock(int blocknum, dfs_block *b): read one filesystem block from the disk, storing it in dfs_block b. Remember that one filesystem block can span multiple numbers of physical blocks. Return DFS_FAIL on failure, and the number of bytes read on success. This fails if the filesystem is not open.
- int DfsWriteBlock(int blocknum, dfs_block *b): write one filesystem block to the disk. Return DFS_FAIL on failure, and the number of bytes written on success. This fails if the filesystem is not open.

You are encouraged to write any additional functions that may help your implementation efforts.

3. (15 points) **Implement the following inode-based functions in the DFS driver**. Note that all of these functions fail if the file system in memory is not valid.
    - int DfsInodeFilenameExists(char *filename): look through all the inuse inodes for the given filename. If the filename is found, return the handle of the inode. If it is not found, return DFS_FAIL.
    - int DfsInodeOpen(char *filename): search the list of all inuse inodes for the specified filename. If the filename exists, return the handle of the inode. If it does not, allocate a new inode for this filename and return its handle. Return DFS_FAIL on failure.
    - int DfsInodeDelete(int handle): de-allocate any data blocks used by this inode, including the indirect addressing block if necessary, then mark the inode as no longer in use. Return DFS_FAIL on failure, and DFS_SUCCESS on success.
    - int DfsInodeReadBytes(int handle, void *mem, int start_byte, int num_bytes): read num_bytes from the file represented by the inode handle, starting at virtual byte start_byte, copying the data to the address pointed to by mem. Return DFS_FAIL on failure, and the number of bytes read on success.
    - int DfsInodeWriteBytes(int handle, void *mem, int start_byte, int num_bytes): write num_bytes from the memory pointed to by mem to the file represented by the inode handle, starting at virtual byte start_byte. Note that if you are only writing part of a given file system block, you'll need to read that block from the disk first. Return DFS_FAIL on failure and the number of bytes written on success.
    - int DfsInodeFilesize(int handle): return the size of an inode's file. This is defined as the maximum virtual byte number that has been written to the inode thus far. Return DFS_FAIL on failure.
    - int DfsInodeAllocateVirtualBlock(int handle, int virtual_blocknum): allocate a new filesystem block for the given inode, storing its blocknumber at index virtual_blocknumber in the translation table. If the virtual_blocknumber resides in the indirect address space, and there is not an allocated indirect addressing table, allocate it. Return DFS_FAIL on failure, and the newly allocated file system block number on success.
    - int DfsInodeTranslateVirtualToFilesys(int handle, int virtual_blocknum): translate the virtual_blocknum to the corresponding file system block using the inode identified by handle. Return DFS_FAIL on failure.

4. (10 points) **Write test code in ostests.c to test the full file system driver**. You should test writing non-block-aligned sets of bytes, writing a file large enough to use indirect addressing, and verify that any written bytes are persistent between runs of the simulator.

5. (15 points) **Implement the following file-based API**. These functions should use the file system driver functions found in dfs.c. You will need to implement a file descriptor structure that stores relevant information about an open file such as the filename, the inode handle, the current position in the file, an end-of-file flag, and the mode. Note that the same inode can be open in single file descriptor. Also, the maximum number of bytes that can be read or written at any time by the file functions is 4096 bytes. All of these functions should only allow the process that opened a given file to use the open file descriptor.

- int FileOpen(char *filename, char *mode): open the given filename with one of two possible modes: "r", "w". If opening the file in "w" mode, and the file already exists, the inode should first be deleted and then reopened. Return FILE_FAIL on failure, and the handle of a file descriptor on success. You can use dstrncmp function (misc.c) to compare strings.
- int FileClose(int handle): close the given file descriptor handle. Return FILE_FAIL on failure, and FILE_SUCCESS on success.
- int FileRead(int handle, void *mem, int num_bytes): read num_bytes from the open file descriptor identified by handle. Return FILE_FAIL on failure or upon reaching end of file, and the number of bytes read on success. If end of file is reached, the end-of-file flag in the file descriptor should be set.
- int FileWrite(int handle, void *mem, int num_bytes): write num_bytes to the open file descriptor identified by handle. If the file is opened with mode="r", then return failure. Return FILE_FAIL on failure, and the number of bytes written on success.
- int FileSeek(int handle, int num_bytes, int from_where): seek num_bytes within the file descriptor identified by handle, from the location specified by from_where. There are three possible values for from_where: FILE_SEEK_CUR (seek relative to the current position), FILE_SEEK_SET (seek relative to the beginning of the file), and FILE_SEEK_END (seek relative to the end of the file).
- int FileDelete(char *filename): delete the file specified by filename. Return FILE_FAIL on failure, and FILE_SUCCESS on success.

6. (10 points) **Write a user application to test your file API**. Demonstrate that you can open a file, write bytes to it, close it, re-open it, and read the same bytes back.

7. (20 points) **Write a disk buffer cache in your DFS filesystem driver**. Rename your DfsReadBlock funtion to DfsReadBlockUncached, and your DfsWriteBlock function to DfsWriteBlockUncached. You may have to use the Queue API in queue.h to implement an LFU replacement policy with an empty_slots queue and a full_slots queue (You can use alternative ways). Your buffer cache should have a total of 16 slots. You can use the following data-structure for buffer-cache node (include/os/dfs.h).
    - inUse
    - dirty: To indicate the block is dirty and must be written to disk during eviction
    - dfs block number: Indicating the dfs block cached by this buffer-cache node
    - count: Number of file operations
    - Link* link: To move this node around empty and full queue.

   Implement the following functions in dfs.c:
    - int DfsReadBlock(int blocknum, dfs_block *b): checks the buffer cache for blocknum, and loads it from the disk into the buffer cache if it is not found. It then copies the bytes from the buffer copy into the dfs_block b.
    - int DfsWriteBlock(int blocknum, dfs_block *b): checks the buffer cache for blocknum, and reads it from the disk into the buffer cache if it is not found. It then writes to the buffer cache copy of the block, and marks the block as dirty.
    - int DfsCacheHit(int blocknum): checks the cache for the given blocknum. Returns DFS_FAIL if blocknum is not found, and the handle to the buffer cache slot on success.
    - int DfsCacheAllocateSlot(int blocknum): allocate a buffer slot for the given filesystem block number. If an empty slot is not available, the least frequently used block should be evicted. If this evicted block is marked as dirty, then it must be written back to the disk.
    - int DfsCacheFlush(): move through all the full slots in the buffer, writing any dirty blocks to the disk and then adding the buffer slot back to the list of available empty slots. Returns DSF_FAIL on failure and DFS_SUCCESS on success. This function is primarily used when the operating system exits. You will need to call it from your DfsFileSystemClose function.

   Test your buffer cache with the file test functions written in the previous problem, using debugging statements in the OS to clearly demonstrate that your caching is working properly.

Turning in your solution

You should make sure that your `lab4/` directory contains all of the files needed to build your project source code. You should include a README file that explains:

- how to build your solution,
- anything unusual about your solution that the TA should know,
- **NOTE: please mention the files that you have modified for each of the questions**,
- and **a list of any external sources referenced while working on your solution**.

DO NOT INCLUDE OBJECT FILES in your submission! In other words, make sure you run "make clean" in all of your application directories, and in the os directory. Every file in the turnin directory that could be generated automatically by the DLX compiler or assembler will result in a 5-point deduction from your over all project grade for this lab.

When you are ready to submit your solution, change to the directory containing the `lab4` directory and execute the following command:

```
turnin -c ece595 -p lab4 lab4
```

---

ece595@ecn.purdue.edu