I got three responses to my call for comments on how you would go about understanding the Xinu file system code. I would like to thank those who have responded and I would like to encourage others to take part in such discussions. You still have time to respond till Thursday, March 2. I find these responses quite thoughtful. I also appreciate that you have discussed your work experiences in similar scenarios. – Kothari

---

Response 1

You have asked us to start considering how we would go about understanding the file system. I would approach it as I do code written by others at work, in a top down manner working from the general to the specific. If I was lucky enough that the original writer still worked for the company and was available for questions, I would go to him or her asking for an overview of the code, which files contain relevant code and generally how the code is organized in those file, similar to the question in class the other day on which files contain the file system code.

Sometimes the person who wrote the code is unavailable, and no one else is available with knowledge of that code. So I would do a file search for key words on the functionality I am interested in, like file open, write or read to get me to a starting place. But we already know from class that the file system code is in the files prefixed with fs, and that the related disk code is in files starting with ds, so I am already ahead of the game.

I would open the files starting with fs first and read the comments first and look at any comments in the source files, and note the coding standards. For example, I knew I was interested in learning how to create files so I looked file starting with fs that relates to open. Finding none, I then went to files prefixed with ds that contain the disk system files and noticed the file named dsopen. Opening that file, I read the comments and read over the code, noticing that there were certain files referenced in the header file: conf.h, kernel.h, disk.h, lfile.h and dir.h. Since header contain the declarations of external functions and type defs and #defines, etc, I would look at those. In particular the header lfile.h gives me some information about the layout of the files on the disk, including the flblk structure. I would note that lfile.h also references header files: iblock.h and file.h, so I would look at those.

At this point, I will be coming up with questions. For example, when I first read the comment at the top of the lfile.h, I was a little confused at first about the referencing of IBlocks. It made sense to me that since each disk block of 512 bytes is divided into 8 iblocks each of which are 64 bytes long that the offset of iblock “k” within its disk block is computed by the formula 64 * remainder(k, 8) (see the fourth sentence of the comment in lfile.h). However, that seems to conflict with the formula given in the previous sentence, which states that IBlocks are referenced relative to 0 so the disk block address of iblock “k” is given by truncate(k/8) + 1. These seem like two different formulas being used to compute the same thing, the offset of an iblock from the beginning of its disk block. So I would seek clarification of that if I had access to the writer of the program or someone who has knowledge of this code.

Once I looked at those seven header files included directly or indirectly in the dsopen.c file, I would then look at the code to see what it does and what methods it references and find which files they are located in. I would keep digging down into each of the called methods to see what they do, and iterate until I became comfortable with the code. If I had time and was really curious about the workings of a method, I might step through it in a debugger seeing how a typical run through the functionality behaves in the code.

As computer engineering/computer science students, we already know what features the file system is trying to implement. Often we do not know that in the real world so we would need to gather domain knowledge about how that area of code is supposed to behave. That way when looking at it for purposes of making changes, we have a better idea of what changes are needed to achieve the right functionality.

Response 2

Below are some thoughts about how one goes about reading/understanding a piece of software:

A simple way to articulate the software development process would be the following:
1. A programmer is given a problem
2. The programmer develops a model to solve the problem
3. The programmer implements that model in code

This is the case for all types and sizes of programming projects. The thing to realize is that step 2 occurs every time regardless of whether a formal development process was followed or not. This is not to say that all programmers have a UML Static Structure diagram in front of them when they are coding. What it means is that it is impossible to write working code without some sort of idea about what one is attempting to accomplish. A data structure does not consist of random elements, but instead elements that the programmer believes will be useful in accomplishing his/her assigned task. There is nothing to say that the model was not developed just prior to (or during) actually writing the code, or that the model is consistent or complete. Instead I am suggesting that every line is there because the programmer was attempting to implement some sort of model in code.

In order to fully understand a piece of code one has to understand what the underlying model is (or was), or in other words "what was the prior programmer thinking". This may or may not be possible in all cases, especially when code has been maintained by numerous people. In this case, the maintenance programmer will also have some task and some model in mind but the goals will most likely be different. Some of the code which implemented the original model may be changed, and new code could be added. This could obscure for future readers the original model.

The statements above are also an over simplification. For example, I say that "a model" is used where it may often be the case that a section of code (even a single function) may be built from progression of simpler models. It may also be the case that a piece of code is the result of a union of several models e.g. a security model, an efficiency model, a portability model, a functional model etc.

If a person truly understands a piece of code they should be able to answer two questions about every line: what does that line do? and why is it there? In order to answer the first question the person must be able to read the code syntax. In order to answer the second question the person must be able to develop and understand a conceptual model that closely matches what the original programmer used when writing the code.

Response 3

Here are some questions that I would ask myself if in order to understand how the file system works:

- What is the call-down structure of the file operations? I would want to know how these calls are made, which functions are making these file operation calls and what are the parameters of these file operations. I am currently doing a similar task a my job where I am required to implement code for a new flash device to be used with our existing software. I've never implemented code for a flash device. So in order to come up with functions that use the flash device, I looked at previous code that implemented different types of flash chips. I specifically looked at the call-down structure of the functions. Though the previous flash chips are not the same as the chip I am to implement, seeing the call-down structure of the functions hel! ped me understand how each function of the flash device is used. This gives me a good idea of how to implement the code for the new flash device.

In this same way for the file system, the call-down structure can help with how the functions are called, what functions are making the calls, what parameters are involved in the file operations, and what the actual values maybe when making these calls. This gives me a good understanding of where different functions are, if there any utility functions that are used and how these utility functions are organized (are the utility functions in one file or do they span several files?). This helps in the overall understanding of how the file system accomplishes its file operations.

- What kind of file tyeps does the XINU OS support? This would allow me to see if it treats all the files t! he same or does the OS distinguish them in certain way. It would also help me see if the OS would allow the file system to do the same operations on different files or if file operations are limited to certain types of files.
- What is the structure of the file? Are there different structures for each file type or is there a common structure for all file types? This will give me an idea how each file operation handles the different types of files or if they are all handled the same way.