

# Team Defrag

**June 4<sup>th</sup> 2009**

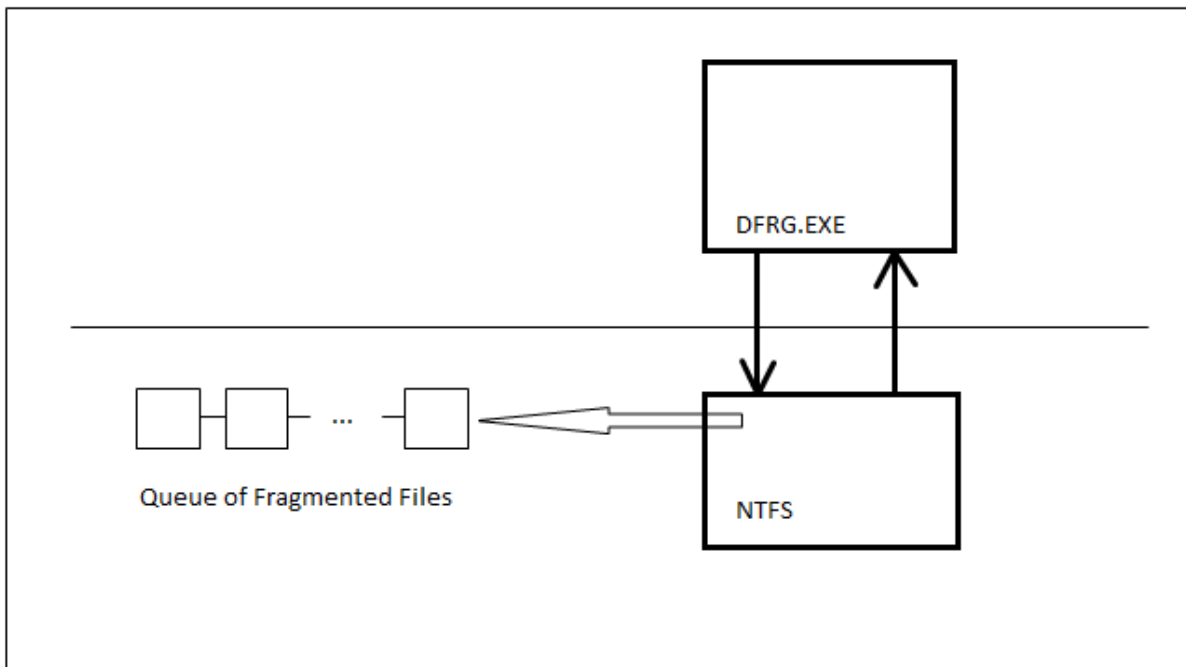
**Victoria Kirst  
Sean Andersen  
Max LaRue**

## Introduction

Our original OS capstone project goal was to introduce an automatic defragmentation of the disk by the kernel instead of relying on a user-level application to fix an inevitable fragmentation problem. With the current practice, disk fragmentation only worsens and worsens as time goes on, with nothing to combat fragmentation until a defrag is performed. We tried to implement a way for the file system to help maintain efficient file storage automatically. Instead of aiming to defrag the entire drive automatically, we defrag recently used files. Any time a file is touched, the NTFS driver checks to see if this is a fragmented file. If so, it schedules a defrag on this file for the future. Having the system defrag on a file-by-file basis is a simple way to help maintain efficient storage space usage without having too much of a strain on the performance while a file is being defragmented in the background.

The resulting project did not accomplish all of our original goals. We simply ran into too many issues to effectively accomplish every goal in the ten weeks of the quarter, only about 6-7 of which we worked on the project. As a result, our final product is more like a proof of concept than a polished, releasable work. We describe the details of what we did accomplish for our project in the next section.

## Project Details and Implementation



### Current Version

Our objective was to defrag files on a per-file basis, prioritized by how often a file is used. To do this, we set up an interface between the NTFS and a user-level application DFRG.EXE. The kernel-level component of our project creates and maintains a work item queue of fragmented files. The user-level app DFRG.EXE requests files from the file system's work item queue and completes the defragmentation of the file. The app then reports back to the file system when it is finished, and the file system updates the queue accordingly. When DFRG.EXE doesn't have a file from the queue to defragment, it works on compressing free space to reduce space fragmentation.

Our product is a roughly a sketch of what we would implement in practice. For example, DFRG.EXE would ideally run in the background of the computer silently with low priority. In our current version, it is a command-line application that runs manually and takes a lot of CPU. The limitations of our project are discussed later in the paper.

## Modification to NTFS

At the kernel-level, we check the fragmentation of a file whenever the file is closed<sup>1</sup>. If the file is in any way fragmented (i.e. if its data is contained in more than one extent), then we add it to our work item queue as a file in need of defragmentation.

Our work item queue is implemented as a linked list of work item nodes. A node is composed of a file reference number and a file status. A file in the queue can be in three states: ItemReady, ItemInProgress, or ItemError. Files are initially added to the queue with a status of ItemReady. This status signifies that the item is in need of defragmentation. When the user level app has requested a file to defrag, the queue item's state changes to ItemInProgress. If the user level app has tried to defragment a file but could not, the item remains in the queue with the status of ItemError.

We added two custom control codes to be used by the DeviceIoControl function to communicate between our user-level app and NTFS: FSCTL\_TD\_GET\_FILE\_TO\_DEFRAG and FSCTL\_TD\_MODIFY\_QUEUE\_ITEM.

The control code FSCTL\_TD\_GET\_FILE\_TO\_DEFRAG is used to get the first item from the work item queue. This does not remove the item off the queue, but instead looks for and returns a copy of the file reference number for the first item in the queue that is marked ItemReady. Queue items that are in progress or that have been marked as problematic files are ignored. If there is no item in the queue that's ready to be defragged, it returns a status code saying such.

The control code FSCTL\_TD\_MODIFY\_QUEUE\_ITEM is used to tell NTFS what to do with a particular item in the queue. This operation requires a file reference number and a desired status. If our desired status for a file reference number is "ItemCompleted", NTFS will then dequeue the item after this call. Otherwise it will simply update the item in the list with the status specified.

## User-level component: DFRG.EXE

Our user-level application waits for a fragged file to be queued in NTFS. It checks for any items in the queue by polling NTFS via a call to DeviceIoControl with our control code FSCTL\_TD\_GET\_FILE\_TO\_DEFRAG. If the queue is not empty, it tries to defrag all files in the queue, then goes back to waiting and polling. While it is waiting, it attempts to "compress" the files on the disk by shifting all file fragments toward the beginning of the disk one cluster at a time.

Once the user-level app received the reference number of the file to defrag, actually defragging the file was done in three steps:

1. Build full path of file from file reference number
2. Find contiguous free space big enough to hold the file
3. Move the file to this free space

We obtain the full path of the file because the operation we were using to move the file, FSCTL\_MOVE\_FILE, required a handle to the file as a parameter, and in order to get a file handle, we needed to supply the full path of the file to the CreateFile function call.

Finding the full path of the file from just its reference number was not a trivial task. From the user level, you can make a call to DeviceIoControl to retrieve the file record segment header for a file given its reference number. The file record segment header includes an offset to the first attribute, which you can use to begin looping through all the attributes for the file. The file name attribute is stored in an attribute with a type code of

---

<sup>1</sup> Originally we checked fragmentation on open, which is perhaps the more intuitive location, but that led to some annoying problems. For example, at one point of our project, a file would be dequeued after it was requested from the user ... but to defragment the file, we would have to open the file again, which queue the file again!

\$FILE\_NAME. However, the file system does not store the full path of the file as an attribute, but rather stores the name of the file or folder and a file reference to the parent directory. To obtain the full directory, then, we have to *build* the path: we first find the name of the file; then we find the file record of the parent directory, find the name attribute of the parent directory, and prepend this name to the path being built; then we continue travelling up parent links until we reach the root directory and prepend the volume name to our path.

Once we had the full path of the file, we to find a place to put it. We attempt to maintain a contiguous chunk of free space and keep track of the cluster number that begins our contiguous free space chunk. If there is not enough space in our contiguous free space chunk, we scan the entire volume for contiguous space big enough for our file.

Finally, we move the file to the free space by a call to DeviceIoControl with the control code, FSCTL\_MOVE\_FILE. If this succeeds, we issue the control code FSCTL\_TD\_MODIFY\_QUEUE\_ITEM and asked NTFS to remove the file from the work item queue. On failure, we ask NTFS to flag the file as ItemError so we know not to move try to move that file in the future.

### Creating Free Space

When there are no files left to defragment, DFRG.EXE begins shifting the fragments of the disk toward the front of the disk. Starting immediately after the system files, DFRG.EXE scans the bitmap for any used cluster. When it finds a used cluster, it shifts the cluster of data over as close to the beginning of the disk as it can.

To this random cluster in the disk, we have to figure out the file to which the cluster of data belongs. NTFS does not provide an easy mechanism through which to do this; there are no built-in mappings from LCNs to file references, so we have to do a reverse MFT lookup ourselves. Starting from the beginning of the MFT, we open each file (which involves transforming the file reference number to a full path to a file handle, as described earlier) and look at the LCNs for all the extents of the file. When we get the extent of the file that lies on this LCN, we compute the VCN of this cluster in the file and use this information to move the cluster.

### Issues We Faced

At a high level, our project task seemed fairly straightforward. But throughout the quarter we ran into numerous road blocks that forced us to rethink our design. We tried many different ideas and learned a lot of what doesn't work, and tried to apply what we learned at each iteration of our design.

### Original Design: Two Threads in Kernel

Our original goal was to have two threads to work on different pieces of our system that initialize when NTFS starts up. One thread would ask our queue of fragmented files for a file to work on and would attempt to place that file contiguously in a location marked by the other thread. This other thread would analyze different pieces of the volume and decide where it wanted to try to create contiguous free space.

The Win32 API has several useful routines to access file information and move clusters around the disk from user mode. Specifically, a program can call the DeviceIoControl function with a FSCTL code to specify what functions to execute. DeviceIoControl calls down into the kernel and figures out which driver to send the request to. The Windows Defragger calls DeviceIoControl from user mode to defrag the disk. Our idea was to make essentially the same API calls at the kernel level in these two threads – e.g. call the kernel-level functions that the DeviceIoControl calls resolve to.

However, we ran into significant problems trying to accomplish this within NTFS. During this process an IRP is created that holds the IO information related to this activity, and the IRP is passed into the NTFS functions that do the desired work. The NTFS functions for defragmentation were designed to be called in this manner, so we could not call these functions correctly from within NTFS unless we created an IRP with the correct information.

At the onset of our project we tried for many days to create an IRP by hand within NTFS by filling in the fields of the IRP. This was never successful, and Mark recommended creating our own control code for each of the threads that would start the threads up and use the IRP passed in to call the NTFS defragmentation functions. Each time we wanted to call the NTFS defragmentation functions we had to edit the IRP to reflect the state of what we wanted to do and we were unsuccessful with this as well. As time passed we felt we were wasting valuable days working on uninteresting issues; instead of gaining experience modifying the file system and learning more about Win2K, we were spending days experimenting fruitlessly over the same few parts of code. This direction did not seem very promising, so we went back to the drawing board to revisit our design.

We eventually settled on our current implementation, which has an important user-level component that communicates with the file system to do defragmentation. Creating a user-level application to interact with the file system had its own share of issues and roadblocks, and ultimately we had to make a lot of design compromises to get our project completed by the end of the quarter.

## Design Flaws

Our current implementation is a long shot away from being perfect. Throughout the quarter, we developed our code with the mantra of, "Get it working first, then get it working quickly and intelligently later." Unfortunately, sometimes "later" never came, and thus there are several areas that we implemented quickly just to get parts working with the intention of improving them down the road, but we ended up running out of time.

One such area is how the user level program and the NTFS communicate. Currently the user level program polls NTFS asking for files to defrag, and if there are no files the user level program tries defrag space, and when neither files nor space are in need of defrag, the program sleeps. We wanted to instead use an event and have NTFS signal when there was a file to defrag so we did not waste cycles polling. This was a low priority item because we wanted to try and spend time on more pressing issues so we did not end up fixing it.

Another imperfect area is the speed in which we accomplish our defrag. The user level program is not very efficient and we simply ran into too many issues to get it working super fast. The *Implementation* section above details exactly what we ended up doing. We also do not defrag every file that is opened and fragmented. Sometimes there are errors getting information about a file or moving the file so we simply ignore that file. This works pretty well because in our experimentation we were able to move 99% of files. In a perfect world we could have tried to move these error prone files multiple times, or also try to move them right when NTFS starts. Our implementation is somewhat of a proof of concept because we accomplish the tasks we set out to but not efficiently.

Certainly we had trouble with maintaining contiguous free space as well. Finding and creating free space can be a very expensive operation computationally. If we always shifted all clusters to the beginning of the disk before we defragmented a file, defragmentation would be an incredibly costly operation. But scanning the entire bitmap for free space every time to place a file is not very efficient either, and we become stuck if there is no contiguous space large enough to place a defragged file.

Our free space making and finding technique is a best effort on a time crunch, and there are plenty of flaws. Our cluster shifting technique is slow and inefficient. Every time we move a cluster, we have to potentially scan the entire MFT, open every file, and loop through the LCN for every extent in every file entry. Worse, we move one cluster at a time, even if a file is large and contiguous, so we unnecessarily repeat the same costly look for each cluster of the file.

Even more problems come up when considering the dynamic nature of files in the disk. Files are often being created and deleted and modified. We keep track of what we think is the beginning of a large contiguous chunk of free space, but at any point in time someone could add files into the space we created, rendering our free

space marker out of date. Currently we have no way of checking for this and instead only detect it once a file is unable to be moved into our supposedly contiguous free space.

## NTFS Complaints

Our complaints with NTFS are directly related to the pitfalls we faced throughout the quarter. We had to make significant revisions to our initial idea for our project because of the unfriendly design of many of the NTFS functions. The `fsctl.c` file contains all these functions that take an `Irp` and an `IrpContext`, which we found to be incredibly difficult to create on our own. These are the only arguments passed into many of the functions and they are passed along from function call to function call. It encapsulates much of the essential state of the current routine (e.g. the parameters, the return buffers, etc.). In some ways, this is great: a function can have access to all these data fields and information as long as it gets passed the `Irp` and `IrpContext` as arguments. But trying to create or even modify one of these structs was very frustrating and complicated. There's just too much state to maintain in the object, and too little direction provided on how to create a proper `Irp` and `IrpContext`. In the end, we couldn't figure out how to create our own `Irp` and `IrpContext`. Since just about every function necessary for defragmentation required use of these parameters, ultimately we couldn't do our entire project in the operating system and had to resort to a user-level compromise.

Writing the user-level application shed light on other parts of the Windows API which we thought were unfortunate. Something in particular that seemed disappointingly difficult to do was getting information about a file from its file record number. There is an FSCTL called `FSCTL_GET_NTFS_FILE_RECORD`, which gives the file record header for the file with the given record number. This gives some basic information, including an offset to the first attribute for the file. From there, though, you have to dig through the bytes manually to retrieve the attributes of the file. This was a tedious process from both a computational and coding perspective.

Since it was still possible to obtain the information of a file from the user level through the Windows API, it is unclear why there were not functions or control codes made to abstract some of these details from the programmer. It seems natural that one would want to obtain a handle to a file from its record number, but as described in previous sections, this was a surprisingly involved task. Perhaps it is because this is unavoidably a computationally expensive task and making it difficult to do indirectly discourages programmers from doing it. Nonetheless, had this been simplified, we would have saved days of researching, coding, and debugging.

## Future Work

The 6 or so weeks we had to work on this project was not enough. We ran out of time and could not successfully implement many of our ideas efficiently, and there are plenty of areas for improvement.

We would start by polishing up our existing code. For example, currently `DFRG.EXE` asks NTFS for a file to defrag, and if there are no fragmented files in the NTFS queue the user level program sleeps for a while and then asks NTFS again. We could improve this polling model by adding a signal that would wake up the thread asking for a fragmented file which would eliminate wasted processing.

In our current version we have our system hardcoded to some extent. `DFRG.EXE` only works on the C volume. NTFS adds all files to the fragment queue if the file has more than one fragment. It would be nice if this was tunable in the sense that someone could change that threshold to only add files if they have more fragments, or only add files if the file has a certain fragmentation to size ratio. This would not be too difficult and would not take long to accomplish.

The queue of fragmented files is just a FIFO queue. Files are defragged in the order they are added to the queue. It would be beneficial if the files that are opened much more frequently were serviced before files that are opened only once. We could improve performance by instead creating a priority queue that assigns the highest priority to files that have been accessed the most frequently.

As described earlier, maintaining free space is an issue which our implementation leaves much to be desired. This is such a large and involved task that a thorough discussion on issue alone could take another paper, so instead we will discuss one idea of improving our current implementation of pushing file fragments into a contiguous group at the beginning of the disk. A method creating free space that was examined involved adding a level of indirection to the master file table and the disk drivers which would contain a mapping from the logical cluster numbers stored in the MFT to the actual clusters on disk. This would allow us to modify the on-disk clusters freely, updating only our added level of mapping instead of the MFT. This means we could (however naively) push all of the files together, leaving the free space contiguous on disk, without having to worry about going through the standard routines to move files, or worry about working with the MFT, which has proven to be tedious. Some drawbacks to this would be the inability for the file system to tell where the files are actually located on disk, trivializing system programs such as the default disk defragmenter program. This would also require a much higher level of integration into the file system than the previous method.

Accomplishing our goals inside of NTFS was not easy as we discovered. If we had much more time we could modify NTFS enough to successfully do this entire project within NTFS. This would reduce a lot of wasted time switching between user and kernel mode. In order to do this we would have to address the issues we mentioned above and we would have to modify the existing FSCTL defrag functions to be called from within NTFS.

## Conclusion

Modifying the file system is never a trivial task. Planning changes to the file system at a high level is almost always far easier than implementing the changes. In our initial attempts at the project, we were very naïve and did not think of many intricate details that complicate our basic goals. But despite our complaints and frustrations, these details made the project very challenging and entertaining.

In the end, we were able to develop and deploy a rudimentary prototype of our system. The tangibles of the project – the user level app, our changes to NTFS – are imperfect, and it's always somewhat frustrating to live with a product with so many known limitations. But we feel what's more important than the coding was the experience and knowledge that we are able to walk away with. We started out with a fuzzy, high-level idea of what we wanted to accomplish, and by the final weeks of the quarter, we developed a solid idea of how to actually implement a fairly solid working version of this project given enough time. While our project goals were not completely met, we feel successful because we learned very valuable information about Windows, NTFS, and systems programming as a whole.