It's <u>Still</u> Loading?

Designing an Efficient File System

By Scott Bilas

Abstract

Every year, engineers are handed more and more content to churn through their game engines, often with the files numbering in the thousands and filling up multiple CD's. Designing a file system to efficiently deal with this kind of quantity will take some careful planning. It will have a significant impact on memory footprint, load times, and general game play chunkiness. Plus, during development it will affect the overall production process, the frustration level of the team, and the tightness of the feature-to-bug-to-fix loop. This paper describes the requirements of a "good" file system and then details how to design and build one. Topics covered include: resource packages, proper use of memory mapping, integrating filters and compression, building tools for packaging, and production process gotchas that proper planning can easily solve.

Audience

This paper is intended for engineers and technical producers and assumes some basic familiarity with how virtual memory works. It includes discussion of memory-mapped file methods that, while not an uncommon feature on most platforms, will use terminology and functions specific to the Win32 API. However, developers for other operating systems should be able to port these concepts over fairly easily.

What is a File System?

A *file system*, for the purposes of this paper, includes:

1. *Executable code*

   Find resources such as PSD's or MP3's, and load or stream them into physical memory. Provide methods to iterate over available files, and do searches.

2. *Data content*

   This includes the directory structure and naming conventions for the resources, plus any other process involved in content creation.

The file system includes more than just bits on a disk – it's important to also put careful thought and planning into the accompanying process in constructing, maintaining, and organizing those bits. Note that "those bits" can be anything that the game requires to execute. Typically it will be a pile of categorized types such as sound (MP3) files, bitmap (PSD) files, or other game engine-specific types like meshes, models, and configuration data. All of these must be somehow requested from within the game, found somewhere on the computer's hard drive or CD (or even from the network), and then made available for the engine to use.

The Guinea Pigs

This paper draws extensively from my own experience in designing and building the file systems for Gabriel Knight 3 from Sierra Studios (shipped November 1999), and Dungeon Siege from Gas Powered Games (slated to ship holiday 2000). For GK3, the set of content consisted of:

- Over 36,000 unique resources – texture maps, models, scene BSP's, scripts, various configuration files, etc.
- Over 2 gigabytes of raw data, about 800 megabytes in final compressed form plus the large in-game movies.
- Data was spread across 3 CD's plus a minimum of 150MB installation size.
- Resources were stored in a context-dependent path tree for artist convenience, but all files uniquely named – no relative pathing required to access a resource.
- File system 100% based on memory-mapped files.

Dungeon Siege has very different content requirements than GK3, but still uses the same low-level design for its file system. It uses relative pathing rather than unique naming for its resources.

Four Basic Requirements

These are my basic requirements for a "good" file system:

1. *Easy to Use*
2. *Fast and Efficient*
3. *Use Package Files*
4. *Solid Build Process*

I'll cover these requirements in the next four sections.

Requirement 1: Easy to Use

This is a top priority – it must be easy to use for the *entire* team. First it must be easy for engineers, meaning that it should be simple and safe to code, especially for junior level developers. It should use familiar file access conventions. Everybody is familiar with the `fopen()`, `fread()`, and `fclose()` style functions. Cater to this. Provide an API compatible with this convention that is based on using safe handles opened by one function, closed by another, and passed to all other API functions as a dereferencing parameter. Then provide a second API that goes directly to the metal using file mapping (more on this later).

It should also be easy for content developers to use. They should be able to easily drop in new resources and modify existing ones. Do not require them to jump through any hoops or have to talk to an engineer to do their work. They should be able to operate independently. Also they need to be able to easily verify that their resources work properly, preferably just by running the game to see what happens. It would be ideal if this can be done without restarting the game. In my experience, the fastest development scenario goes like this:

> *A content developer runs the game. They see a resource that needs modification or are intent on creating new resources – a new level, a new monster, adjusting a texture map, etc. They alt-tab away from the game to their development environment, which may be a 3D modeling program, a paint program, or an external game-specific tool like a level editor. They update the content from their separate tool and save or export it out to their local drive, then alt-tab back into the game. They tell the game to "rescan". It rebuilds any indexes affected by the overriding local directory, and the content developer can immediately see the results in the game. This is what we did on Gabriel Knight 3 and are continuing to do on Dungeon Siege. It works well.*

It would be even more efficient if the editor were built into the game itself. Changes could then be made and the results seen in near real-time. Though this solution is potentially quite a bit more work to implement, I believe it's the ultimate goal in content creation – the game is the editor is the content is the game. Content developers who work with the web are accustomed to this sort of thing already.

Requirement 2: Fast and Efficient

As the title of this paper implies, gamers don't enjoy waiting. The infamous yet strangely standardized level-is-loading progress bar or flashing text is a low point of every gaming experience. A properly designed file system can be fast and efficient in the code by staying close to the hardware, and vastly improve performance over a naïve implementation. Modern virtual memory operating systems give engineers great opportunities to do this, generally via memory-mapped files.

The code must support these file access scenarios:

1. *Block*

   This is probably the most common access method – it's a one-shot block read used to initialize something (such as a bitmap). Typically the entire resource is read into memory, processed, and then closed.

2. *Random*

   Pure random access. Used for randomly reading from a very large file, perhaps for pulling in parts of a world map on demand.

3. *Streaming*

   Like random except a serial pattern. Typically used for playback of sounds, animations, and movies.

Requirement 3: Use Package Files

A package file is a huge file containing multiple resources, possibly compressed or otherwise post-processed. Examples of package file types in commercial products are WAD, HOG, ZIP, BRN, and TANK. Use of package files is commonplace in games today, but there still seems to be a percentage of developers (even those published by big names) that continue to ship games with thousands of raw BMP and WAV files installed to the hard drive. Perhaps this section can help convince developers that it's worth the extra effort to use package files – they have some major advantages:

1. *Single static file handle*

   Only a single file handle is used per package file, and it's left open all the time. Finding and opening a file using the operating system can be relatively expensive, especially on NT systems where file security must be checked on each `CreateFile()` call.

2. *Maximize disc space usage efficiency*

   Small files will always take up the minimum cluster size on the drive, which can be wasteful in percentages of installed size. I measured a waste of around 10% in one recently released game that stored over 3200 files in a single directory on an 8K cluster size FAT32 drive. Results will vary with cluster size and average resource size. Note that this is not a problem on CD's, where files are packed end-to-end.

3. *Static indexing*

   OS file systems are general purpose. They support finding, reading, writing, renaming, adding, and deleting of files. With the obvious exceptions of save-game and local resource caching, most of the time your game will simply need to find and read data from a fixed data set. Using the OS to look up and find files is very slow because the underlying code and data structures are designed to be general purpose. There are a few options to tune the system to boost performance a bit, but you can still do a much faster job of this on your own using indexes based on hash tables or trees contained completely in memory. *This is a big win*.

4. *Custom ordering of data*

   Games typically have predictable access patterns, especially if they have a *level* concept – resources will often be read in the same order every time that level is loaded. It's easy to physically order this data in the sequence it's going to be requested by the game. This will minimize the very expensive necessity of seeking the disk's read/write head, plus it will take advantage of natural locality of reference for improved cache usage. As you pack the data into your package file you can store it using this optimum ordering.

On the other hand, installing thousands of individual files to a disk, besides taking much longer than one big package file, will probably end up scattering them all over the place. You have absolutely no control over the physical location on the disk that the operating system chooses to store its files and will spend a lot of time waiting for the disk head to seek. Note that this is not a problem on CD's, where you can order the physical locations of the files however you like.

5. *Versioning is easy*

   Tracking the version of a single or a small set of package files is very easy to do. The file will have a custom header – store version information in it. This makes it simple to run patches and do expansion packs. Checking the versions of and patching potentially hundreds of individual resources would be much more difficult to coordinate.

6. *Requires a build process*

   Something has to build the package file – we'll call this the *packager*. This packager has the unique job of having to access every single file the game will ever use. This is a Good Thing, as it provides a single funnel through which all resource data must pass. This is covered more in the fourth requirement.

7. *More professional*

   The installation of a single 100M package file is going to proceed far more smoothly and appear more professional than 10,000 individual files. It's easier for the end user to manage as well. Plus the source data will be a little more difficult to rip from the game, assuming this is a concern for you.

Requirement 4: Solid Build Process

A packager tool will need to be constructed to build the package files. This tool will "compile and link" all the resources together. As mentioned previously, it will have all the data in the game pass through it on a regular basis – as often as builds are performed. Take advantage of this opportunity. Data can be verified, compressed, or modified on its way into the package file. Depending on the game, this can be a critical necessity.

Verification of the files as they go in is fairly easy and can happen in two ways – locally and globally. Local verification will occur on an individual file that can be tested on its own. Text configuration files and script source files would fall into this category. They can be tested for proper syntax. Note that verification of binary files (such as a mesh file) is probably not necessary, as the tool that generated them can guarantee that the formatting is correct. But in the case of bitmaps, the packager can certainly test to make sure that texture maps are power-of-two in each direction, or have a particular bit depth or extra channel of information. Checking for zero-length files is also a good idea.

Global verification occurs on a higher level, and deals with the organization of the data. Naming conventions can be checked on the resources as they are found. Or say a particular scene in the game asks for a set of models and animations. Normally this would only be tested on demand at runtime (i.e. when the level is loaded). The packager can easily test this as it goes. Obviously more advanced testing can happen as well, and the needs will certainly depend on the game.

The build tool can also support *filtering* – modification of the data as it goes in. As it's pulling in the data, it can:

- Convert Photoshop format bitmap files to a more efficient internal game surface format.
- LZ-compress non-WAV data.
- Precompile scripts into object form.

The build process can benefit from having such a tool in other ways. Building the package files will be a major event that all the content developers can synchronize to, similar to an engineering EXE build. It also provides a historical snapshot of the state of the game, which is critical during testing. Also the build process can be streamlined – one big batch file can build the executable and the content.

Six-Part Solution

My solution to meet these four requirements consists of six parts. These should apply to nearly any type of game. It will be especially effective with games where frame-to-frame content requirements greatly exceed available memory. By this I mean games where content is constantly pouring through the game on demand during game play rather than just on a level load. This is the solution that shipped with Gabriel Knight 3, and an improved variation of this will ship with Dungeon Siege.

Solution Part 1: Organized Network Resources

This part of the solution is talking about content process. Put all game resources out on the network and have the development team run exclusively from net data. Source control systems make this very easy – Visual SourceSafe offers *shadow directories* which will maintain an exact copy of the content tree on a separate drive anywhere on the network. Also, put the latest build of the EXE up on the network and have the content developers run directly from this as well. Back up older builds and then copy the new build over the old one. This ensures that as new builds go up everyone is always running the latest.

Running everything from the network can save hundreds of hours of wasted debugging effort. By guaranteeing that everybody is in sync, it nearly eliminates problems of someone running an older build, or of having mismatched, incorrect, or outdated content, or of having local files that should have been deleted. The shadow directory on the network is always the latest data, the correct data, and the exact data that everybody else will be running. In addition, it's the data that the packager will be using, so it's also the official content that will ship on the retail CD. This should cut down on the always-mysterious "it works fine on my machine" phenomenon.

This will require that the game is able to run with local overriding resources. A developer will check out a piece of content from the source control system, edit it, and then want to test it out in the game before checking it in. The game should be able to accommodate this by overriding network resources with local resources. Also set up the source control client to automatically delete the local resource upon checking it in, so that the game goes back to running that data off the network. Now obviously putting all of this data on the network is going to be slower. The game will take longer to load and run. You can minimize this problem by adding code to the game's file system that allows it to dynamically change the location it gets its data from as it is running (see Requirement 1) which will keep the developer from needing to restart the game. This is fairly easy to code, and is covered in Solution Part 3.

Note that there is an important issue to be aware of when running with network resources. When the game opens a resource for reading, it will be locked to the rest of the world for updates. This means that anyone trying to update the file by checking in to the source control system will succeed the checkin and fail the shadow directory update. This will cause a synchronization problem between the source control database and the game resources. Check for this case every couple weeks or so with a recursive project diff. From within the game's file system, it can be made a non-issue by writing a little bit of extra code that will detect opening of a remote file resource and copy it to a local memory buffer as soon as it's opened, then closes it immediately. This is simple to implement and is a lifesaver for game resource integrity.

Set this process in stone long before production begins. Choose sensible naming conventions, and enforce them from code if possible (in the exporter, or the packager).

Solution Part 2: Simple File Access API

The engineering file system will require two basic API's – one for old-fashioned, but familiar, `fopen()` / `fread()` / `fclose()` style access, and another for new-fangled memory-mapped file access.

The `FileHandle` API would provide `Open()`, `Read()`, and `Close()` functions. This API would be familiar to all programmers, and easy to port existing code to use. Underneath, it would be based on memory-mapped files, which can easily emulate ordinary file access with a file offset and `memcpy()`. The `MemHandle` API would provide `Map()`, `GetData()`, and `Close()` functions. The `GetData()` function would return a simple pointer to the data, which is directly mapped to the resource on disk using the OS's memory-mapped file system. This is much simpler to use though less familiar than the `FileHandle` API. Here is a sample interface:

```
class IFileMgr
{
public:

// File i/o functions - standard read-only file pointer data access.

    virtual FileHandle  Open    ( const char* name ) = 0;
    virtual bool        Close   ( FileHandle file ) = 0;
    virtual bool        Read    ( FileHandle file, void* out, int size ) = 0;
    virtual bool        ReadLine( FileHandle file, string& out ) = 0;
    virtual bool        Seek    ( FileHandle file, int offset, eSeek origin ) = 0;
    virtual int         GetPos  ( FileHandle file ) = 0;
    virtual int         GetSize ( FileHandle file ) = 0;

// Memory mapped i/o functions - read-only access via memory pointer.

    virtual MemHandle   Map    ( FileHandle file, int offset, int size ) = 0;
    virtual bool        Close  ( MemHandle mem ) = 0;
    virtual int         GetSize( MemHandle mem ) = 0;
    virtual const void* GetData( MemHandle mem ) = 0;
};
```

In this system, a `FileHandle` and a `MemHandle` would be similar to standard system handle types like `HANDLE` – different types but both 4 bytes in size for function passing efficiency.

The purpose of the `IFileMgr` interface is to abstract the file system. Requirement 1 says that we must run from network resources. This requires a system that works with raw files found on local or remote paths, and supports local overriding of remote data. Requirement 3 says to use package files. This requires a system that knows how to pull resources from one or more package files. So part of this solution is to implement three separate versions of the `IFileMgr` interface – one that implements the path file management, another that implements the package file management, and a third that abstracts a collection of `IFileMgr` derivatives so that both file systems can be used simultaneously.

The indexing structures of the implementations are affected by naming conventions. There are probably two main methods of generally accessing files in a system. First is *unique naming*, which is what Gabriel Knight 3 uses. In this system, the data set is treated as a flat array of resources with no path information. A resource is accessed by name – `Open( "wood.bmp" )` – and this resource may exist anywhere on the set of available path trees. In other words, the resources may exist scattered across a set of paths or they may all exist in a single directory – it makes no difference to this system. The advantages are that it's very simple to use. All files are simply referenced by their name with no need to know where they are located. It's easy to implement this system for packaged files, and packaged file index lookups are very fast (binary search or hash is fine). Disadvantages are that it requires indexing of the paths in development builds, which is very slow. Assuming that your resources are scattered across the network, divided up for artist convenience, the game still needs to be able to find them via a global search, which requires indexing on the fly. This takes some time to implement. Also, there can be problems with duplicate filenames existing on separate paths, though this can be taken care of upon package file construction. And because files are located by name rather than by path, naming and location conventions are almost impossible to enforce from code, which can encourage rather haphazard shotgun style resource storage. This can be a real problem for a multi-CD game when it comes time to separate out the resources.

Another method of accessing files is *relative unique naming*, and this is what Dungeon Siege uses. The data set is treated like a directory tree, where files are referenced by name plus path information. When opening a file, the relative path to a resource must always be specified – `Open( "art\maps\wood.bmp" )`. The advantages of this system are that no indexing is required in development builds. You can simply give the system a root for its files and it will prefix all file-open requests with that path. Duplicate names are not possible because the OS prevents it, and it's very easy to enforce file location standards through code. For example, you may decide to require that all texture art necessary for a particular level be located with the convention `levels\<levelname>\maps`. This is easy to check from code – if the file does not exist in that directory, then it can't be found and the game fails to load the level, printing out the appropriate error messages. Strict requirements on names and locations like this make game content easier to manage, especially if the data must be separated in order to split the content across multiple CD's. The main disadvantage of this system is that implementing a package file system is more difficult, given that you must implement a directory tree rather than a simple flat array.

I've found that the second (relative unique) method of naming is far better and probably more common as well. The more complicated package file indexing is well worth it.

Solution Part 3: Path-Based File System

As mentioned in part 2 of the solution, we need a path file manager to use as an interim system for development. Because we want to be able to swap new and modified resources in and out while the game is running as fast and easily as possible, we'll be working with the network resources directly. It's far too inconvenient and slow to have to rebuild the package files just to change a few resources.

The path-based file system pulls its data from an ordinary directory structure, either locally or out on the network. Configure it by handing it a root directory or set of directories to work from, and it will process file open requests to pull data from those directories. It should support absolute pathing for testing convenience, meaning that for development builds, you should be able to hand it a path like `c:\temp\test.bmp` and it will use that file directly. In addition, it needs to support the concept of "override" paths, where the developer can choose a set of local paths that will override the ordinary game content paths. This is necessary so that the developer can check out a file to a local working directory and have the game use that file rather than its older version out on the network.

For a relative unique naming system, the path file manager is simple to implement. First it needs to check to see if the filename is absolute, and if so, attempt to open it directly. Otherwise it should iterate over its available root paths, starting with the overrides, prefixing each root to the relative unique name and attempting to open the file until it finds it. Given this system, using new or changed resources without restarting the game is simple and automatic.

For a unique naming system, the path file manager can be more complicated to implement. A file open attempt could take forever if it has to iterate over the entire network tree each time, so have it index the paths as it goes. As the path file manager checks a particular directory for a file, it can remember what other files are in there for future queries. This can significantly speed up file searches and amortize the cost of indexing a little if only done on demand. Allowing the use of new or changed resources in the system will probably require throwing out the entire index and starting over, but this is certainly faster than reloading the entire game. The local override paths can be left unindexed to minimize the need for rebuilding the index.

The path file manager, though intended as an interim development system, can also be shipped with the game and left disabled by default. This will allow end-users to customize their data by simply turning it on and building override files. Depending on the game, this could be a very good thing, enabling users to create their own themes, new units, modified AI, etc. This of course should be disabled for multiplayer games to discourage cheating a little.

Solution Part 4: Flexible Package File Format

The package file format will be binary and will require at the least a header, an index, and the content data itself. The header will need information about the package file itself: its original name and size, a checksum, required EXE version to use, and build date and number. This is needed to check for corruption or user modification, and to support updates to content and new expansion packs.

It's easy to come up with a simple proprietary format for the package file, but I recommend using the Win32 Portable Executable file format. This is the standard format that all Windows images are stored (EXE's, DLL's, OCX's, etc.). Use this format and store the package in a section, then add a `.rsrc` section with a `VERSIONINFO` resource. This is fairly easy to do and has the significant advantage of allowing developers to view the properties of the package without needing to run a tool to query it. Using the Windows explorer, a PE-format file with a `VERSIONINFO` resource can be viewed by right clicking on it, selecting "Properties", and then viewing the "Version" tab of the dialog. The version resources can contain, along with version information, name-value pairs with whatever you like in it. This can help debug content problems more quickly in providing a fast way to verify which version a package is, plus being able to easily see any other important attributes.

For the index, the simplest thing to do with a unique naming system is to store the entire set of filenames as a flat array and binary search over them. For a more efficient and cache-friendly system, use a hash table. At minimum the file entry could just be the size and offset within the file that the resource is located. For a relative unique naming system, the data structure will have to be a little more complicated in order to maintain its memory and query efficiency.

This situation can be made even more complicated by adding multi-CD requirements. If the game has any possibility of being spread across multiple CD's, this needs to be designed in from the start. Note that this doesn't mean that the game happens to ship on more than one CD – the term "multi-CD" refers to the game needing to swap CD's in the middle of gameplay. This is very different from having an "install" CD that is only used once on initial game setup, and then a "gameplay" CD that is used the rest of the time. The ability to change CD's while the game is running may require that the game know about all possible resources across all CD's at any time – this makes a global resource index necessary.

There are other multi-CD concerns. Consider the possibility of an *emergency* CD swap. By this I mean that the game is running normally, and someone put a file the game needs on the wrong CD. In order to prevent serious injury, the game should automatically ask the user to swap CD's, read the file it needs, and then swap back. This is a minor annoyance compared to the price of an outright crash. Gabriel Knight 3 put in this feature at considerable expense, and though (happily) it hasn't been needed in the retail build thanks to our QA team, it helped out greatly during the QA process in keeping people from needing to restart the game when files were on the wrong CD's. Looking back, it would have been much better if we had separated out our data set in advance and enforced this from code somehow. By the way, this separation of data can be a major headache, so budget time for it. Balancing minimum install size versus performance in addition to somehow dividing up the remaining data across the leftover space on all the CD's can take a lot of time to figure out.

Finally, the data itself may be stored in a special format – compressed and possibly encrypted. Compression is very likely given how much data modern games are shipping with and how easy it is to implement. The underlying data format at its lowest level should be hidden from client code. The package file manager will need to auto detect the data format and then convert it to raw form (decompress or unencrypt) on demand. Note that MP3 and other pre-compressed files should always be stored in raw format. Low-level data formats should never be specific to a particular file type, but instead use a general algorithm.

Solution Part 5: Package File Builder

The package file builder, or *packager*, is like WinZip or link.exe. Everybody is familiar with this procedure: (a) choose files, (b) do something to them (i.e. compress/order/verify), and (c) pack them end-to-end in the output file. This should be kept as simple as possible. The packager can be built as either a command-line or GUI application. The command-line version is probably best in that it's the easiest to implement, and can be called from a batch or make process to auto-generate packaged files. Be sure to data-drive it with a configuration file (i.e. don't configure it from the command-line). For convenience use XML for configuration – there are lots of freely available XML parsers that do this, including the one built into Internet Explorer[1].

The packager should support filtering the data as it gets copied to the package file. Use a standard plugin-style interface for defining filters so that it's easy to add new filters later. Changing the data will happen on a file-by-file basis but verification can happen either per-file or globally after the available file set is known. The filters should be kept simple to avoid long build times and packager complexity. For instance, you probably don't want to build a MAX-to-BSP generator into the packager. This is probably better kept as a separate tool.

As mentioned previously, data ordering is important to minimize the seek time required to access a particular resource in a package. Resources that are located next to each other will

[1] More information is available on XML from MSDN at http://msdn.microsoft.com/xml/.

immediately take advantage of the hardware and OS's automatic look-ahead caching, which is usually multi-megabyte in size. This pre-cached data is thrown away if not used, so this is a very worthwhile optimization with respect to game load times. The game's file system can be modified so that it logs an entry in a resource journal every time a file is opened. This journal can be fed back into the packager on the next build run to set the order for the files to be stored in the data section. An exact ordering is not too important – just try to get as many chronologically related resources as close together as possible.

Support for compression is becoming a necessity these days. A CD's typical capacity of 650MB isn't quite the mountain of storage it used to be. In addition, reading the compressed data from a CD and then decompressing it in memory is often faster than reading the uncompressed data, especially if it happens to fit in the look-ahead cache. Adding compression support is easy: there are a couple excellent compression libraries out there that work well, are free to use, and easy to integrate. First is the `zlib` library[1], which offers a high compression ratio, supports streaming of data, and decompresses fairly quickly. Another compression library is the lesser-known `LZO`[2], which has a good compression ratio, does not support streaming, but has a stupidly fast decompression rate (the author claims it can be as fast as 1/3 the speed of `memcpy`). On Gabriel Knight 3 we used `zlib` for compressing streamed resources like animations, and `LZO` for everything else. Dungeon Siege will probably do the same thing.

Solution Part 6: Package-Based File System

This is the center of the file system: the package file manager. It will get used most heavily in the retail build and must be optimized. Some important decisions must be made up front regarding this system. Will it deal with more than one package file at a time? If so, how will the packages override each other in case they contain files with the same names? Will there be the possibility of adding new packages to the system on the fly? For example, a real-time strategy game that stores all the data necessary for a map in a single package file may want the option of auto-downloading the package while in the multiplayer lobby. On the other hand, an adventure game will never need to be modified on the fly – its data set is completely known in advance, assuming the game is not extendible (and they typically aren't). The requirements will be very specific to the game type.

The file system must be efficient. Given the possibility of multiple simultaneous package files, iterating through each of them to find an individual file could end up costing a lot of time, not to mention thrashing the cache and using more memory than necessary. It will be faster and more memory efficient to build a separate file manager that exists as a singleton and maintains a master index of all the available resources – this is what Dungeon Siege uses. The master index must be rebuilt each time the available set of package files changes, but this is fairly quick. Iterating and searching through the available resources must be fast, as the game will be doing it often. After finding the file entry by name, this is one possible data structure to use for dereferencing to the final location:

---

[1] The `zlib` library is available from http://www.cdrom.com/pub/infozip/zlib/.
[2] The `LZO` library is available from http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html.

```
struct FileEntry
{
    std::string m_Name;          // name of the file
    DirEntry*   m_Parent;        // parent directory
    int         m_OwnerIndex;    // which package file is it in?
    int         m_DataOffset;    // offset within the file that the data starts at
    DWORD       m_FileFlags;     // bitfield set for whether file is compressed, etc.
};
```

For the get-rich-quick among us, there is a very old and super-simple working sample of a package file system and packager called FastFile in the Platform SDK[1].

The package file system should exclusively use memory-mapped files. But what are those?

Memory-Mapped Files

A memory-mapped file is built from three things: a file (opened with `CreateFile`), a file map (attached with `CreateFileMapping`), and one or more mapped views (created with `MapViewOfFile`). A file is opened normally, preferably read-only, and a file map is attached to it. The file map is simply a memory structure that views can be attached to. This intermediate step seems to be necessary purely to support shared memory maps for inter-process communication, which is an entirely different topic. The next step is to map views of the file.

A memory-mapped view is a region of virtual address space "backed" by the file. It is built upon a *section object*[2], a low-level kernel object in the OS's memory manager. Pages of virtual memory are directly mapped onto a file in 4K chunks for x86 platforms. Many views can be mapped into the same file, and they may overlap. The only requirement is that there must be enough virtual address space to hold them. The file may be of any size (terabytes if you wish) but the sum total of all the virtual address space used by the process and the memory mapped views must be less than 2G. In practice the actual limit is far lower than this and totally dependent on what the user is already running. This won't be a problem though because we're going to create our views into the package file on demand and destroy them when they are longer needed. Also they will only take up the minimum size required to access the resource. Realistically we won't be using more than 10M at a time, peaking at perhaps 50M if the level loader has a lot of recursion in it – it's the simultaneous view usage that's important. Anyway this is not going to be a problem for us.

Accessing a resource from the file works like this: (a) look up the name in the index, and get its size and offset, (b) call `MapViewOfFile()` using the given size and offset of where it's located in the file[3], and then (c) use the returned `const void*` pointer to access the data directly. This is certainly convenient, but aside from this, why use memory-mapped files? They're fast and efficient, and ideal for read-only package files. Files mapped as read-only are backed by the file itself, not volatile memory. Old pages that must be dumped from physical memory can

---

[1] This is available from MSDN at
http://msdn.microsoft.com/library/devprods/vs6/visualc/vcsample/vcsmpfastfile.htm.
[2] For more information on section objects, see Inside Windows NT (2nd Ed.) by David Solomon.
[3] This must be aligned to the system allocation granularity – for more information, see the MSDN help for `MapViewOfFile()` at http://msdn.microsoft.com/library/psdk/winbase/filemap_8p9h.htm.

simply be thrown away, rather than swapped out to a system page file (which is necessary for buffers allocated with `malloc()`, incurring an expensive write operation). The memory is managed by the system – you don't need to worry about whether a particular resource is taking up memory space or not, or implementing your own resource caching system. The OS does it all for you. Lazy evaluation is the key here. Mapping a view only reserves contiguous address space; it does not take up any physical memory. Accessing a page that has not been assigned physical memory is called a *page fault*. File chunks are only brought into memory ("paged in") on demand. This means that data is only read in when needed. And when it's no longer needed, it no longer takes up *any* resources. This has more of an effect on streaming resources that are kept open than anything else.

Using a memory-mapped file eliminates a lot of layers of code. Rather than using `fread()`, which does its own buffering of calls to `ReadFile()`, which in turn sits on top of other OS functions and probably does more buffering, you can just dereference a pointer. This goes directly to the memory and is much more efficient. A typical old-fashioned file processing operation goes like this: (a) allocate a temporary buffer, (b) read some data into it, (c) check for read errors and handle, (d) process data, (e) repeat for next data chunk. This entire process and all the associated memory copying goes away with memory-mapped files. You simply have a pointer: just iterate through memory, casting and dereferencing as needed. The OS does all the rest of the work of bringing data into memory, caching it, throwing it away when no longer needed. The code required to process game resources using a memory pointer is far simpler than that based on `ReadFile()`.

Error detection and handling is very different for memory-mapped files than for `ReadFile()`. Once a memory pointer is acquired through the `MapViewOfFile()` (which does set an error code on failure) all work is done through pointer dereferencing. If you go off the end of a mapped view, or attempt to access a dangling or uninitialized pointer, there's a very unsurprising access violation that occurs. But what about a damaged file or dirty CD? This is an *in-page error*, which means that the system was unable to pull a page of data in from disk to resolve the fault. The way to handle these access violations and in-page errors is using Win32 Structured Exception Handling `try` and `catch` blocks.

A top-level SEH handler can convert confusing in-page errors to more reasonable "CD is dirty" errors, but in general, there's really nothing useful you can do about it. Win9x will probably blue-screen and tell the user to check their disc anyway. In practice, the access violations will occur because of programmer error, and cannot really be "handled" generically in a meaningful way – so just let them access violate and treat it like a null pointer dereference. These problems rarely go unnoticed for long. For the others, be nice to the end-user by reformatting them to use less jargon and help out tech support a bit. Beyond that there's not much you can do. Practically speaking, as long as the original `MapViewOfFile()` call succeeds, you can pretty much ignore error handling entirely.

Some Potential Issues

It's not all fun and games in memory-mapping land. It would be nice if they worked as they are documented. In Windows NT and Windows 2000, they work *exactly* as documented, and have no problems. The entire OS is based on memory-mapped files, so this is to be expected. Win9x however is built upon DOS and DOS-style I/O, and according to Microsoft its implementation of memory-mapped files is "emulated" and so certain features are not supported.

There are a lot of accompanying issues to be aware of when using memory-mapped files on a Win9x system. Three important Knowledge Base articles talk about them: Q125713[1] and Q108231[2], and Q242161[3]. There are also some weird things the Win9x memory manager does in order to conserve memory, read about them in "Improving the Performance of Windows 95 Games"[4]. Be sure to read all of these articles. I also found that mapping large files through a Novell network redirector doesn't work for more than 600MB to 800MB of files mapped at a time. This is talking about file size, not total views mapped, and I know it's strange and probably specific to drivers or something on that particular machine, but it's not an issue for a CD game. Win9x is also very sensitive to leaked handles, so have tracking in place to make sure that doesn't happen otherwise it'll be reboots-a-plenty for the whole gang. Also try to keep the total number of open views and files to a minimum.

Despite these issues that sound so scary, memory-mapped files work fine for the purposes of a game's file system. Gabriel Knight 3 did ship with no file system problems, as will Dungeon Siege, and both are 100% built on top of `MapViewOfFile()`.

Conclusion

File systems aren't sexy, but they are a foundation service that every game needs. A file system encompasses more than just the code – it covers everything in the system that has to do with content that goes through a game. Properly designed, it can make the game faster and help it ship on time.

[1] Find this at http://support.microsoft.com/support/kb/articles/Q125/7/13.asp.
[2] Find this at http://support.microsoft.com/support/kb/articles/Q108/2/31.asp.
[3] Find this at http://support.microsoft.com/support/kb/articles/Q242/1/61.asp.
[4] This is available from MSDN at http://msdn.microsoft.com/library/techart/msdn_gameperf.htm.