

or well-known, DLL. In this case, the LoadLibrary call will always resolve to the WINDOWS\SYSTEM directory.

**[0036]** DLLs and other shared components also retain reference count semantics to ensure that a component is not touched unless no running applications refer to it. In practice, only applications from the operating system vendor and the operating system itself have done a good job of obeying this protocol.

**[0037]** As a general rule, it is desired to have a shared object always resolve to the correct component. To provide this functionality it is required to understand the version of a component, or range of versions, that an application is able to function with. Then, when the application is to be run, the present invention should ensure that the component is resolved correctly. It is acceptable, in the present invention, to automate the use of WPFS or other operating system provided capability, if desired. In this case, it is necessary to detect needed components and place them in the local file system. This is more complex than just watching installation, as an installation program will often not install a component if the required one is already there.

**[0038]** It is desired to identify a method to ensure that named objects are also loaded correctly. On the Windows platform, MSVCRT.DLL is a significant culprit within this problem area. If multiple versions of this object are maintained, the aforementioned Registry key can be dynamically changed, allowing the LoadLibrary function to resolve the correct component version. Another reasonable method of ensuring correct component loading is the dynamic editing of a process environment to use a valid search path. This search path will ensure that a local component is resolved before a system wide component. Another possible method for resolution of the correct shared object is through the use of symbolic links. A symbolic link can be made for a shared component, which is resolved at run-time by the computer's file system to the needed component. Finally, the actual open/read/close requests for information from a shared object's file can be intercepted by the present invention and responded to dynamically for the correct version of the file which may exist on the local system or within the invention's subsystems.

**[0039]** Several special forms exist. On the Windows platform, OLE, ODBC, MDAC, . . . as well as a number of other vendor specific components, are written to be shared globally among several or all running processes. In the case of OLE, going as far as sharing data and memory space between separate processes. OLE prevents more than one copy of itself running at a time, as do many of these components. OLE also has many bugs and features requiring a specific version to be loaded for a specific application. In the present invention, an application is able to load whatever version of OLE is required, still enabling the shared semantics with other components using the same version of OLE.

**[0040]** In general, unless specifically configured as such, shared objects should be loaded privately to ensure conflict prevention. Nothing about the method used to allow a component to be loaded privately should prevent it from being unloaded cleanly or correctly loading for another software application, whether being actively managed by the Operating System Guard or not. In addition, if the system crashes it is required to recover from this crash to a clean state, not having overwritten or modified the underlying operating system.

**[0041]** Files

**[0042]** Many applications use data files within the application to store configuration entries or other application data. The present invention provides a virtual file system much like the virtual registry described above. Before the application starts, the present invention can load a list of file system changes, including files to hide and files to add to the virtual environment or files to redirect to another within the virtual environment. Whenever the application accesses or modifies any files, the Operating System Guard checks if the file must be redirected, and if so, in the preferred embodiment redirects the request to a location specified in the Operating System Guard configuration.

**[0043]** If an application tries to create a new file or open an existing file for writing on a user's local drive, the Operating System Guard must ensure that the file is actually created or modified in the redirected location. If the application is reloaded at a later time, this file mapping must be reloaded into the Operating System Guard virtual environment. When the request is to modify an existing file, which resides on a user's local drive, the Operating System Guard must copy the file in question to the redirection point before continuing with the request. The redirected files may not be of the same name as the original file to ensure safe mapping of file paths. In the preferred embodiment, INI files are handled in this way to offer maximum system security while allowing maximum application compatibility.

**[0044]** The present invention is particularly useful for applications delivered over a network. In such implementations it is important to understand that software applications are made of several kinds of data, where the bulk of the files a software application uses are most preferably mounted on a separate logical drive. Configuration, including both file based and registry based, can be user specific and system wide. The application delivery system used should mark each file for which of these types any file is. This information provides hints to the Operating System Guard system to act on appropriately.

**[0045]** Device Drivers

**[0046]** Many applications use device drivers or other operating system level software to implement some of its functions such as hardware support or low level interactions directly with the operating system. In the present invention, the Operating System Guard will provide the capability of dynamically, and as possible privately, adding and removing these components to an application's virtual environment.

**[0047]** Many device drivers are built to be dynamically loadable. If at all possible, it is the preferred embodiment to load all device drivers dynamically. If a device driver requires static load at boot time, the user must be presented with this knowledge before running the application. Once the system has rebooted, the application should continue from where it left off. However, a large percentage of device drivers are not dynamically unloadable. Although it is preferred to dynamically unload the driver, if this cannot be accomplished the driver will be marked for removal on the next reboot, and the user should be made aware of this. If the application is run a second time before the next reboot, the system should remain aware of the presence of the driver and not attempt a second installation, waiting for termination to remark the component removable at next reboot.