

[0026] The loader subsystem 116 of the present invention is used to allow virtual environments to be transferred into and out of the running system. Each of the Virtualization Subsystems is capable of serializing its configuration for the loader 116, and retrieving it through the reverse process. In addition, the loader 116 is capable of staged loading/unloading and combining the results of individual stages into one single environment description.

[0027] Registry and Configuration

[0028] Applications require varying amounts of configuration information to operate properly. Anywhere from zero to thousands of configuration records exist for which an application can read its configuration. On Windows, there are two common places for configuration information, the Windows Registry and system level initialization files win.ini and system.ini. In addition, the \WINDOWS\SYSTEM directory is a common place for applications to write application specific configuration or initialization files. Applications will also use configuration or data files in their local application directories to store additional configuration information. Often this information is difficult to deal with, as it is in a proprietary format. On platforms other than Windows, there is no equivalent of the Registry, but common directories exist for configuration information. X Windows has an app-defaults directory. Macintosh has the System Folder, and other operating systems will have corresponding elements. It is important to note that on most UNIX systems, each individual application 52,54 will most often store its own configuration 152,154 locally, as seen in FIG. 2.

[0029] The present invention, in one embodiment, includes a virtual Windows Registry component, which will provide a full function registry to an application, but prevent modification to the underlying system registry. All keys that an application expects to access will be present, but may only exist in the virtual registry. In this way, the Operating System Guard 100 of the present invention and the Windows Registry form a two-stage process for accessing the registry. If an application needs access to a key, it will query the Registry. The Operating System Guard will respond with the key and its value if it knows it. Otherwise, it will allow the request to pass through to the Windows Registry. If an attempt is made to modify the value, the Operating System Guard will allow the modification to occur to itself only. The next time the application accesses the key, it will be present in the Operating System Guard and the request will not flow through to the real Registry, leaving it untouched.

[0030] The keys that the Operating System Guard uses are specified in three separate sections. These Operating System Guard keys are specified as commands in these sections to modify an existing key, delete the presence of a key, or add a new key to the registry. In this way, the virtual registry can appear exactly as the system intends. This is important as the presence or absence of a key can be as important as the actual value of the key.

[0031] In the preferred embodiment, the Operating System Guard first loads a data file that contains basic registry entries for the application. Then a second data file is loaded that contains the user's preferences. Finally, the Operating System Guard can optionally load a set of keys that include policy items that the user is not allowed to override. The three files load on top of each other with duplicate items in

each file overriding items in the file before it. The first time a user runs an application, the second data file will not exist because there will be no user-specific information, only application defaults. After each session, though, the Operating System Guard will save the user's changes, generating that second data file for use in future sessions.

[0032] Configuration files can be modified in two ways. First, the file can be edited directly by an application. In this scenario, the Operating System Guard File subsystem described below will address the modification made to the file. Second, in the preferred embodiment, an application can call the Windows API family of calls GetProfileString, WriteProfileString, or others to modify these files. In this case, the Operating System Guard of the present invention performs exactly as described above intercepting these calls and servicing them from within.

[0033] Shared Objects

[0034] Many components used by operating systems and running applications are shared across several applications or instances. In general, this is a very good idea. It saves disk space, not requiring many copies of the same file. It also provides the ability for operating system vendors and third parties to create and distribute libraries of commonly used code. On the Windows platform, Dynamic Link Libraries, DLLs, are often shared within and across applications. On other platforms, the problem is the same. On the Macintosh, INITs and other system components are loaded for applications. These components can have many versions, of which only one is used at a time. On UNIX systems, dynamic shared objects, e.g., ".so" library files, are used by applications to speed load time, save disk space, and for other reasons. Many programs use the default "libc.so." However, this library file is typically a symbolic link to some version of itself such as libc.so.3. In practice, this feature has created havoc. These shared components have often gone through revision, with many versions of the same component available to be installed. Application authors have found their software to work with potentially only one or some of the versions of the shared component. Thus, in practice, applications typically install the version they desire, overwriting other present versions. This potentially causes defaults in other applications running on a system.

[0035] On Windows 98, Windows 2000, Microsoft has created the Windows Protected File System (WPFS) to allow system administrators to create a file called XXXX.LOCAL in the base directory of an application, where XXXX is the executable file name without the extension. This causes the Windows Loader to alter its method of resolving path references during LoadLibrary executions. This, however, is not sufficient to completely solve the problem. First, setting up the XXXX file is left to the knowledge of the system administrator, which varies widely. Second, a component version must undergo a rewind back to the original, then install this component in the local directory, and then create the ".LOCAL" file. This is not a straightforward process for any but the most basic components placed in WINDOWS\SYSTEM. Also, this solution does not cover all of the needed functionality. During LoadLibrary, Windows uses different path resolution semantics depending on whether the component was resolved as a result of an explicit or implicit LoadLibrary, and also whether a Registry Key exists indicating that it is a named,