

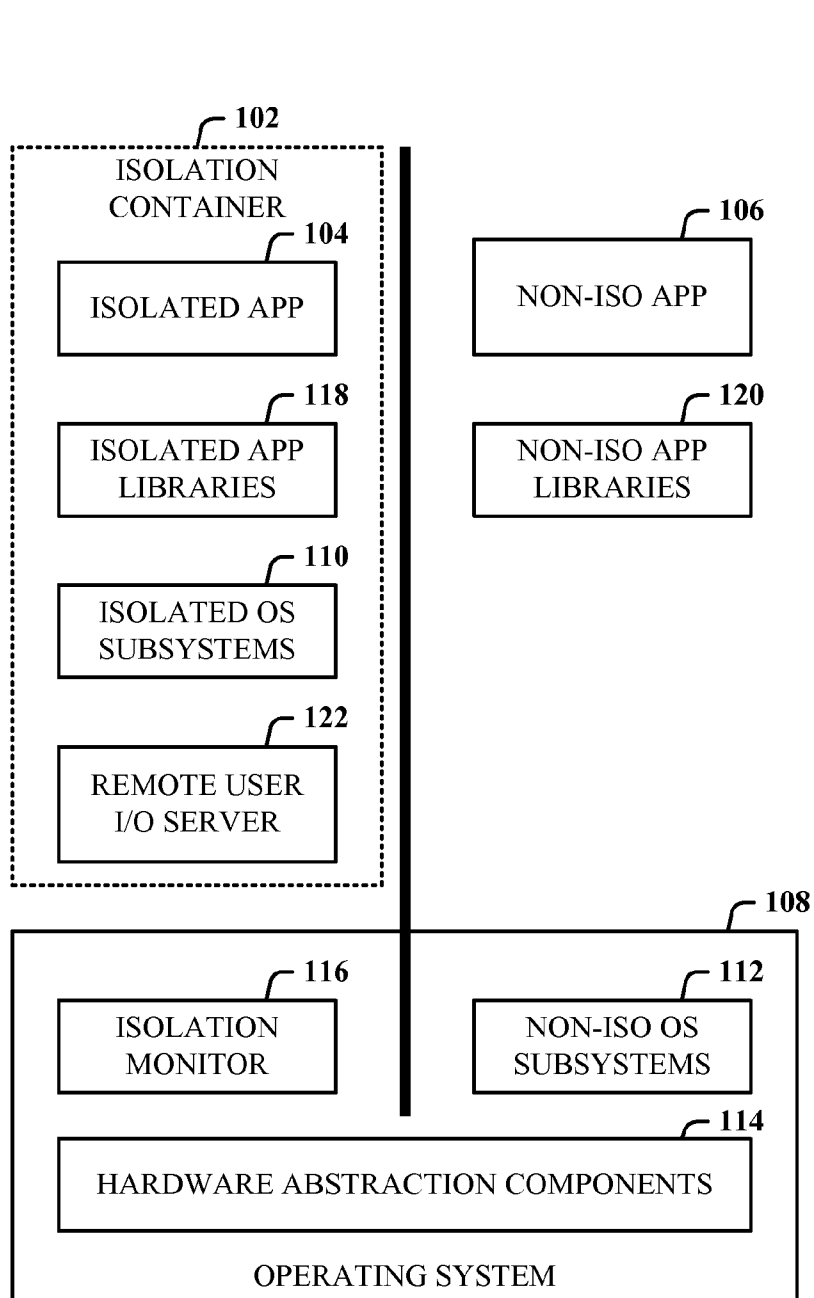


FIG. 1

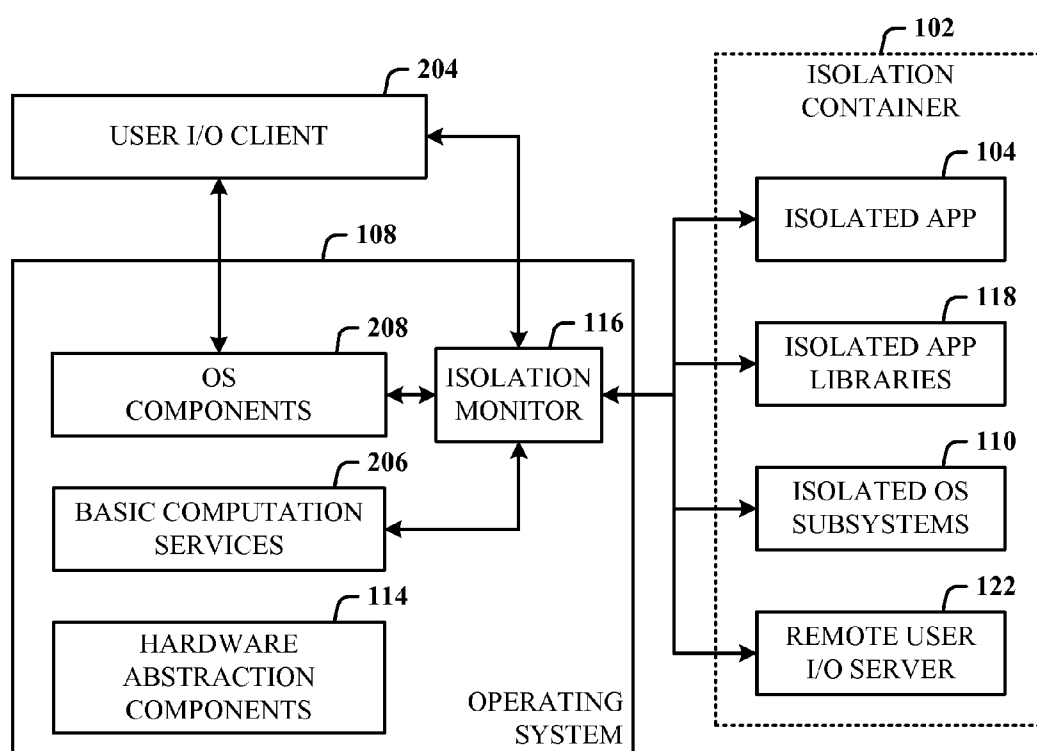


FIG. 2

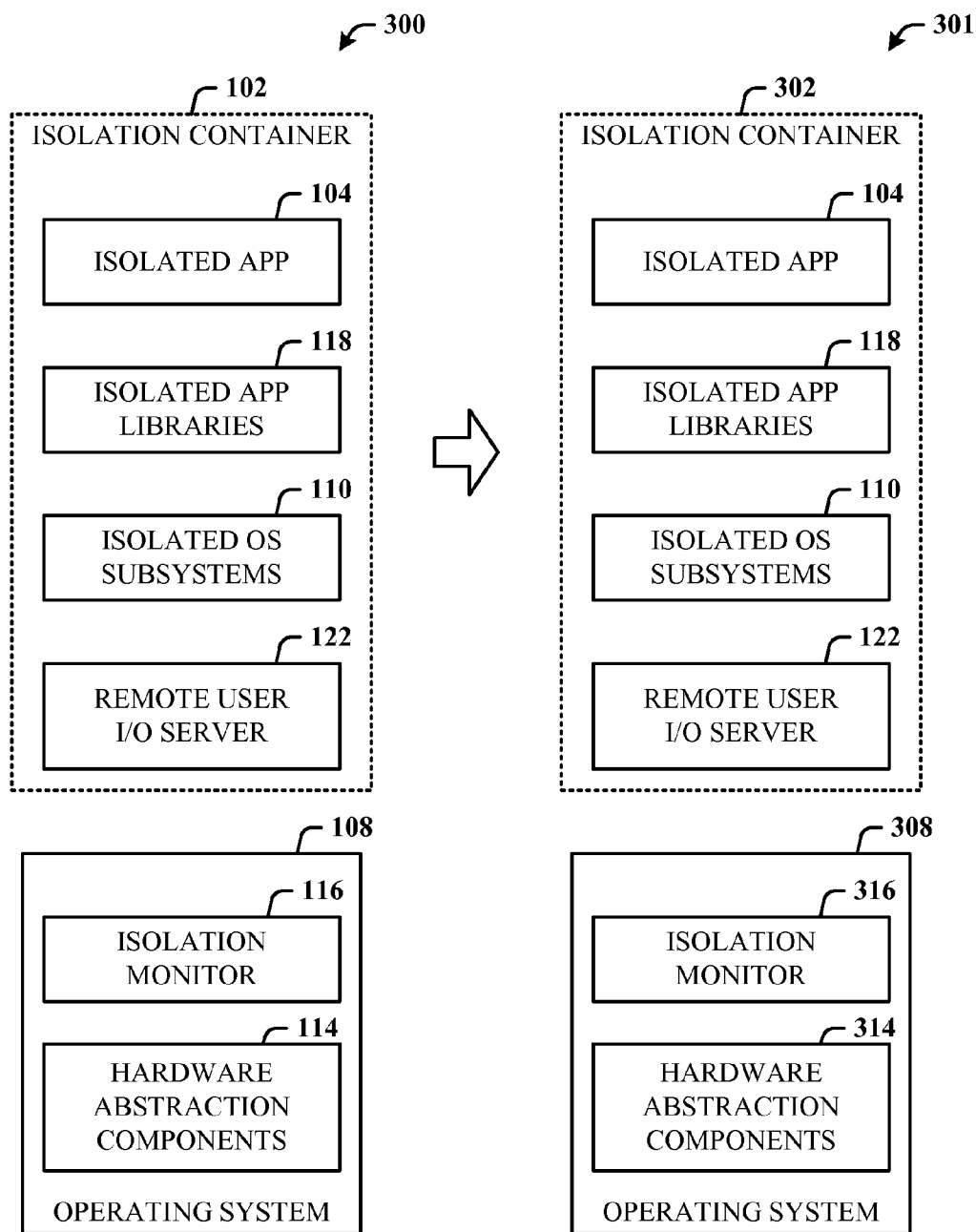


FIG. 3

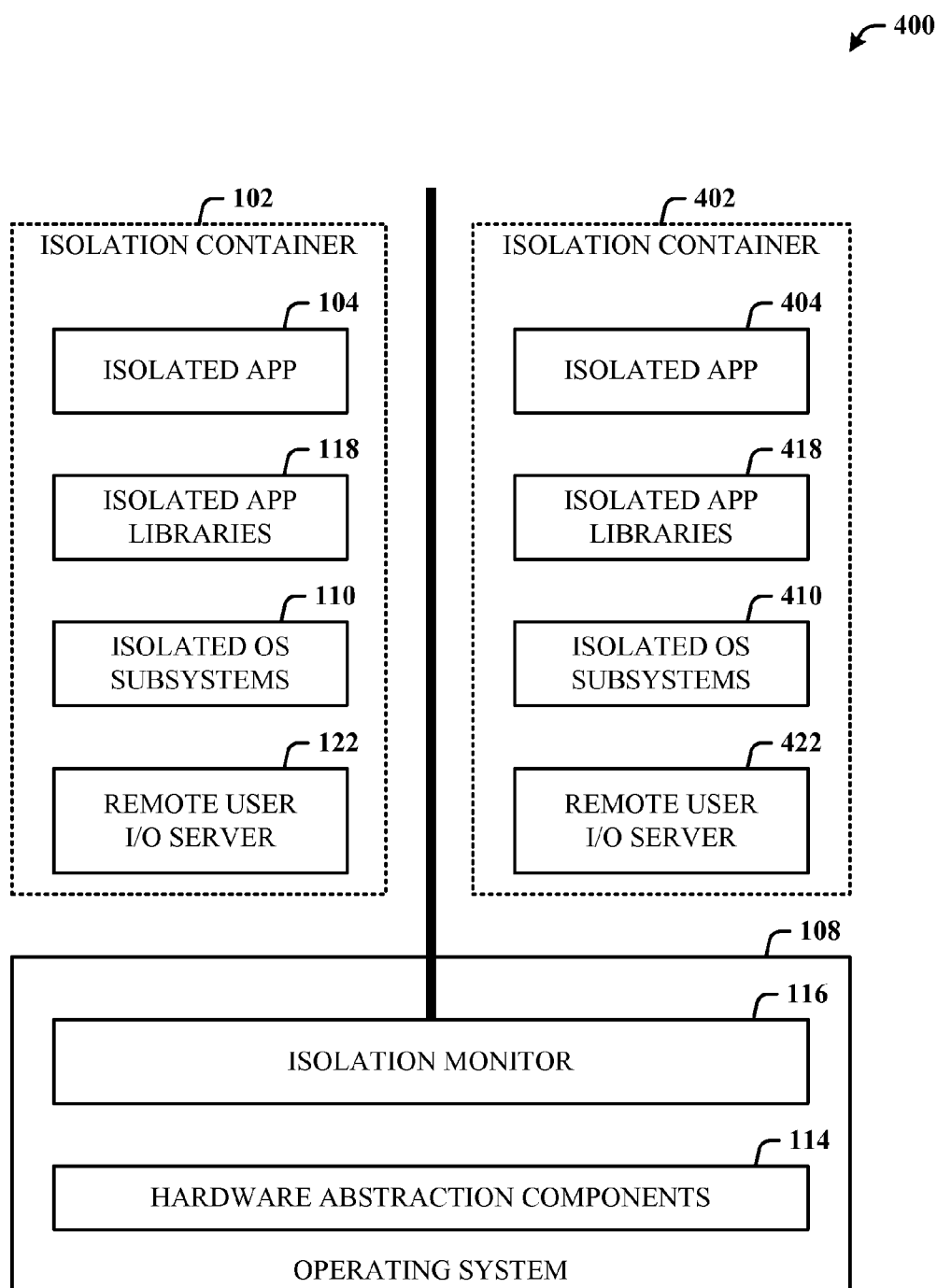
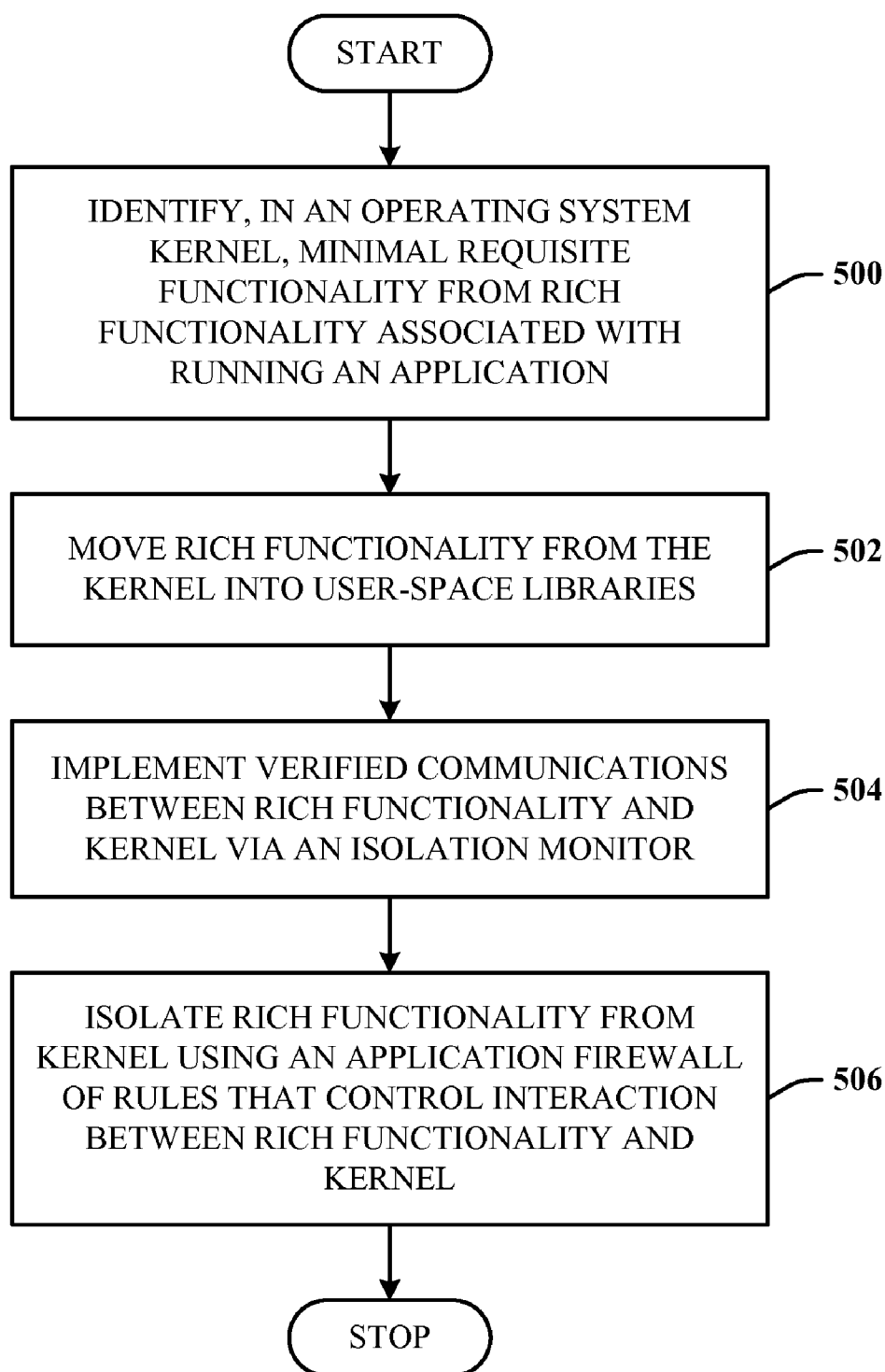
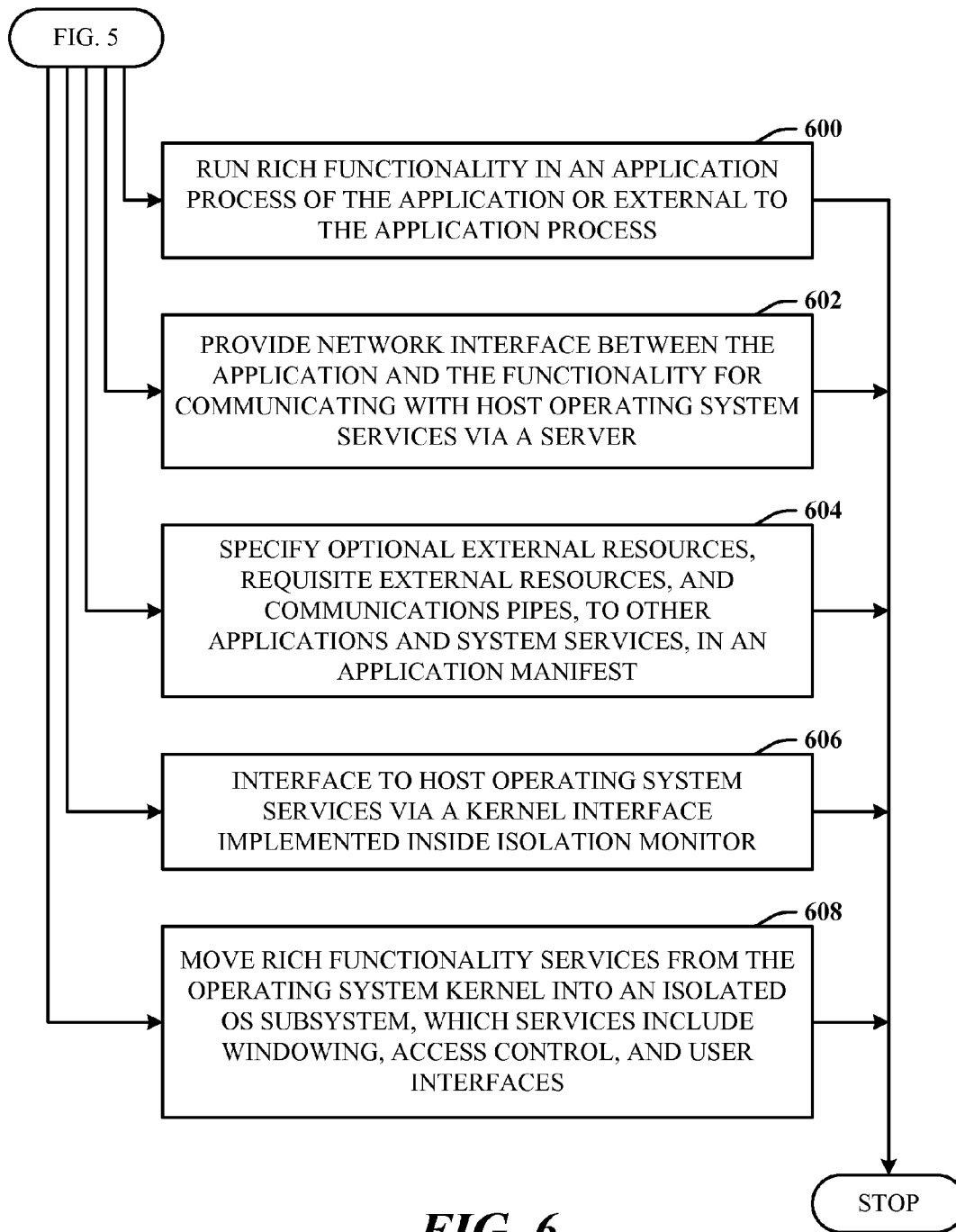


FIG. 4

**FIG. 5**



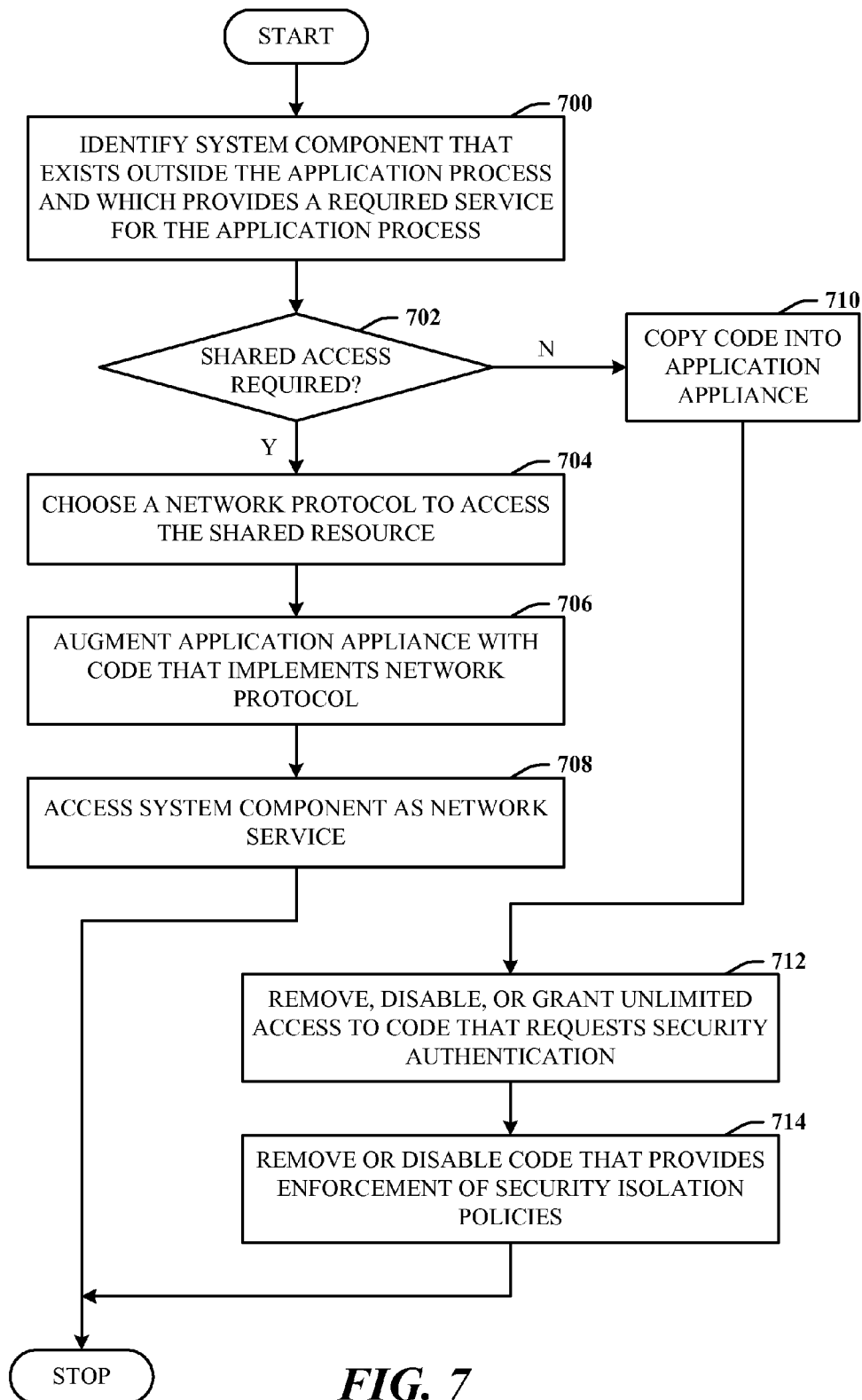


FIG. 7

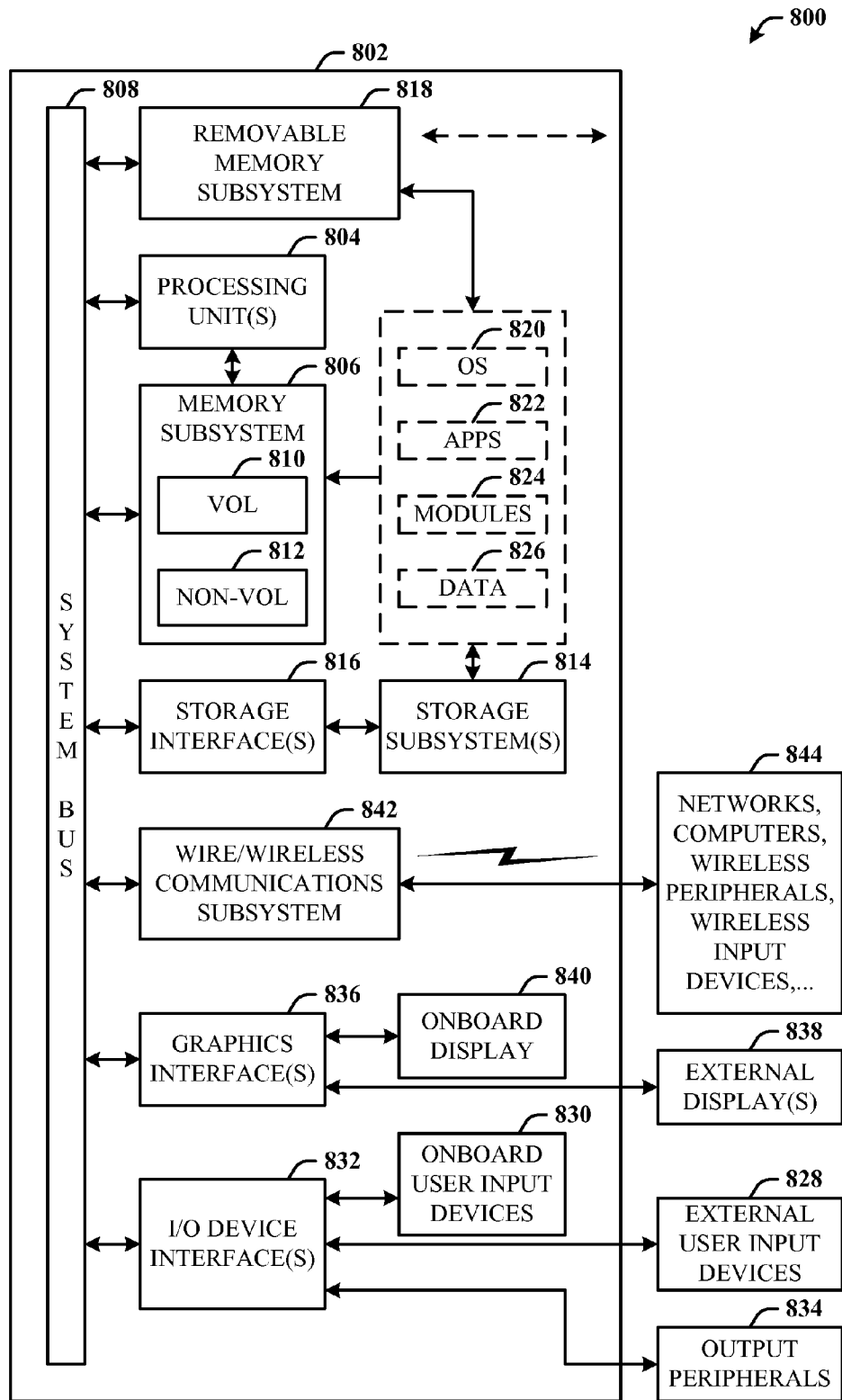


FIG. 8

ULTRA-LOW COST SANDBOXING FOR APPLICATION APPLIANCES

BACKGROUND

[0001] Sandboxing is a security technique for isolating the execution of untested code and untrusted applications. The best prior sandboxing solutions used virtual machines to isolate one application from the rest of the applications on a system. With the application isolated in a virtual machine, the isolated application cannot compromise the state of the system or other applications. The isolated application can also be migrated from one computer to another computer by carrying the entire virtual machine container (both memory and storage). Finally, vendors can create application appliances by bundling an application and the required operating system components into a virtual machine that is distributed to customers.

[0002] Users seldom use isolated virtual machines for security in practice because the machines are too expensive in terms of computer resources because the virtual machines emulate low-level hardware interfaces, thus forcing the isolation container to contain a complete operating system. Furthermore, in common use, only the largest applications (such as server applications) are distributed in virtual machines, again, because the storage resource overheads of including a complete separate copy of the operating system are too high to justify for all but the largest applications.

[0003] Additionally, memory overhead for virtual machines is high because each virtual machine runs a complete (or nearly complete) operating system to abstract virtual hardware (within the virtual machine) to provide the type of environment expect by an application. For example, a standard application expects to run on the abstraction of virtual memory. However, a virtual machine typically provides an abstraction of physical memory with page tables, the mechanisms used by an operating system to create virtual memory. Likewise, an application expects to access a file system, whereas a virtual machine only provides the abstraction of disk blocks. Finally, where an application expects the abstraction of threads of execution, a virtual machine provides instead the hardware abstractions of processors, timers, and interrupts, out of which an operating system creates the abstraction of threads.

SUMMARY

[0004] The following presents a simplified summary in order to provide a basic understanding of some novel embodiments described herein. This summary is not an extensive overview, and it is not intended to identify key/critical elements or to delineate the scope thereof. Its sole purpose is to present some concepts in a simplified form as a prelude to the more detailed description that is presented later.

[0005] The disclosed architecture facilitates the sandboxing of applications by taking core operating system components that normally run outside the application process, and on which the application process depends on to run, and converting these core operating components to run within the application process. To reduce overheads, the architecture takes basic computing services already provided by the host operating system, such as virtual memory and threads, and safely isolates these abstractions for use by the sandbox environment.

[0006] More specifically, new operating system APIs (application program interfaces) are created that include only basic computation services, thus, separating the basic computation services from rich application APIs. The code providing the rich application APIs is moved out of the operating system and into the application isolation environment—the application process (or can be run external to the application process).

[0007] For example, in a Windows™ implementation, the entire Win32 subsystem and the relevant portions of the system registry are copied into the application sandbox so that the sandboxed application runs its own copy of the Win32 subsystem. Since the Win32 subsystem now provides services to only a single application, the Win32 subsystem need not be protected with security checks or other mechanisms, such as placing the Win32 subsystem in its own operating system process, from the application. Rather, the Win32 subsystem can be run in the same process as the application, further reducing the overheads of providing an isolated environment.

[0008] To accomplish this, a remote user I/O server is included in the application process as well. The operating system components, which would normally rely on device drivers to communicate to hardware such as display, keyboard, and mouse, instead use a remote user I/O server, to communicate with remote user I/O devices thereby creating an application appliance. By including all of the external operating system components with the application the standard system call interface can be disabled at the bottom of a process with an ultra-small operating system interface that provides only local compute capability.

[0009] To the accomplishment of the foregoing and related ends, certain illustrative aspects are described herein in connection with the following description and the annexed drawings. These aspects are indicative of the various ways in which the principles disclosed herein can be practiced and all aspects and equivalents thereof are intended to be within the scope of the claimed subject matter. Other advantages and novel features will become apparent from the following detailed description when considered in conjunction with the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] FIG. 1 illustrates a secure application execution system in accordance with the disclosed architecture.

[0011] FIG. 2 illustrates a secure application execution system that utilizes an isolation monitor for communications between the isolated application and the operating system.

[0012] FIG. 3 illustrates future proofing in which an isolated application runs on either a first operating system or a second operating system.

[0013] FIG. 4 illustrates future proofing in which a first isolated application written to run on a first operating system and a second isolated application written to run on a second operating system both run on the same operating system.

[0014] FIG. 5 illustrates a method of creating a secure application execution system in accordance with the disclosed architecture.

[0015] FIG. 6 illustrates further aspects of the method of FIG. 5.

[0016] FIG. 7 illustrates a method of factoring operating system code into components to be used in an application appliance environment.

[0017] FIG. 8 illustrates a block diagram of a computing system that executes application sandboxing in accordance with the disclosed architecture.

DETAILED DESCRIPTION

[0018] Operating systems (OSs) mix basic primitives of computation, such as threads, virtual memory, and file access, with rich APIs (application program interfaces) such as application configuration management, GUI (graphical user interface) services (e.g., the display of windows and direction of keyboard and mouse input to specific windows), and user interfaces components. It is the rich APIs that are desired to be isolated to provide a sandboxed application environment. The disclosed architecture takes the abstractions provided by the host operating system and converts (refactors) these abstractions for use in and by the sandbox environment. Basic APIs are refactored to expose only isolated computation abstractions to code in the sandbox environment. Rich APIs are refactored to run as user-space libraries isolated within the sandbox environment.

[0019] As applied to Microsoft Windows™ OSs, the disclosed architecture refactors a Windows OS and moves much of the functionality required by real applications out of the OS kernel and into user-space libraries. This includes, for example, the complete set of Windows GUI services and the registry—complex components with wide interfaces that traditional Windows implements as shared kernel services. This dramatically reduces the size of the architecture's system-call interface. Behind this narrow interface is a simple and robust TCB (trusted computing base) implementation.

[0020] Running applications according to the architecture provides at least the following benefits: isolation—by moving most of OS functionality out of the TCB, processes are much more robustly isolated than in the OS; migration—removing process' reliance on shared kernel state also allows process images to be easily moved from machine to machine; and, future proofing—each application can incorporate whatever version of the OS libraries it was written against. As the OS evolves, newer applications can be written against new features and use newer libraries on the same machine. This also supports legacy applications.

[0021] This isolation of program state enables the user to start a program and then move the program's running memory image from one device to another, such as from a desktop computer to a laptop computer, from a laptop computer to a mobile phone, from a mobile phone to a server in the cloud, etc. The significant reduction in resources and overhead provided by the disclosed architecture now makes it possible to sandbox every application.

[0022] When applied specifically to a Windows™ operating system environment, the rich operating system components on which the sandboxed application depends are converted to run within the application process. For the Windows implementation, a remote user I/O service is implemented using the remote desktop protocol (RDP) running within the application process as well. The operating system components, which normally rely on device drivers to communicate to hardware such as display, keyboard, and mouse, instead use the RDP server code, thus creating an application appliance. By including all of the external operating system components with the application the standard system call interface can be disabled at the bottom of a process with an ultra-small OS interface that provides only isolated basic compute capability.

[0023] Reference is now made to the drawings, wherein like reference numerals are used to refer to like elements throughout. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding thereof. It may be evident, however, that the novel embodiments can be practiced without these specific details. In other instances, well known structures and devices are shown in block diagram form in order to facilitate a description thereof. The intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the claimed subject matter.

[0024] FIG. 1 illustrates a secure application execution system 100 in accordance with the disclosed architecture. The system 100 includes an isolation container 102 in which an isolated application 104 (denoted Isolated App) runs in isolation from a non-isolated application 106 (denoted NON-ISO App). The isolated application 104 and non-isolated application 106 both run in association with a single operating system (OS) 108. Isolated OS subsystems 110 (denoted Isolated OS subsystems) of the isolation container 102 provide services to the isolated application 104, and non-isolated OS subsystems 112 (denoted NON-ISO OS subsystems) of the OS 108 provide services to the non-isolated application 106. The isolated OS subsystems 110 and non-isolated OS subsystems 112 provide equivalent services to the corresponding isolated application 104 and non-isolated application 106.

[0025] The OS 108 includes hardware abstractions 114 available for both the isolated and non-isolated applications (104 and 106). Additionally, the OS 108 includes an isolation monitor 116 that provides the interface for services from the OS 108 to the isolation container 102. The separation of state related to the isolated application 104 from the state related to the non-isolated application 106 is represented by the black bar that extends between the isolated application 104 and the non-isolated application 106, and down into the OS 108 between the isolation monitor 116 and the non-isolated OS subsystem 112. The system 100 can also include in the isolation container 102 isolated application libraries 118 (denoted Isolated APP Libraries) for the isolated application 104, and non-isolated application libraries 120 (denoted NON-ISO APP Libraries) for the non-isolated application 106. The libraries (118 and 120) expose the services of the OS subsystems (110 and 112) to the respective applications (104 and 106).

[0026] The isolation container 102 may also contain a remote user I/O server 122 which increases the similarity between the isolated OS subsystems 110 and the non-isolated OS subsystems 112 by providing emulations of certain hardware components, such as video displays, keyboards, and mice.

[0027] Note that as illustrated, the isolated OS subsystem 110 and remote user I/O server 122 are external to the isolated application 104; however, it is to be understood that, alternatively, the isolated OS subsystem 110 and remote user I/O server 122 can be part of the isolated application 104.

[0028] Note that as illustrated, the isolation monitor 116 is a distinct, separate component from the other portions of the OS 108; however, it is to be understood that, alternatively, the isolation monitor 116 functionality can be implemented by modifying the other portions of the OS 108 to enable running of the isolating functions of the isolation monitor 116 for the isolated application 104 in addition to running non-isolated functions for the non-isolated application 106.

[0029] The equivalent services may include application configuration management services, GUI services, printer services, and/or audio services, for example. The equivalent services are exposed to the isolated application **104** and non-isolated application **106** accessed either directly or through user-space libraries (**118** and **120**). The libraries (**118** and **120**) are compatible with different versioned isolated and non-isolated applications (**104** and **106**). The operating system **108** further includes hardware abstraction components **114** available for both the isolated and non-isolated applications (**104** and **106**). The operating system **108** includes the isolation monitor **116** which employs a collection of rules that map the approval or denial of requests to access resources to an application manifest.

[0030] The isolated application **104** and the non-isolated application **106** use basic computation services provided by the OS. The basic computation services include one or more of virtual memory management, thread creation, and thread synchronization.

[0031] The manifest defines which resources are optionally available and which resources are available and required for correct execution of the application. The code within the isolation container **102**—including the isolated application **104**, the isolated application libraries **118**, the isolated OS subsystems **110**, and the remote user I/O server **122**—interfaces to the kernel of the operating system **108** through the isolation monitor **116**. The contents of the isolation container **102** may be migrated to a different computing environment by reproducing the address space on the different computing environment and then recreating the threads and other resource handles on the different computing environment using descriptions of those threads and resource handles saved in the address space of the isolation container **102**. In other words, the isolated application can be migrated to a second computing environment by copying the address space of the isolation container or by reading the address space of the isolation container, which isolation container is in a first computing environment.

[0032] FIG. 2 illustrates in more detail the flow of communication through the isolation monitor **116** between the isolation container **102** (also called a sandbox environment) and the operating system **108** and external services such as the display and user I/O client **204**. The isolated application **104**, the isolated application libraries **118**, the isolated OS subsystems **110**, and the remote user I/O server **122** can all request services through the isolation monitor **116**. Services are presented in at least two forms. Services on private virtual memory, threads, private files, and messages to remote services are presented as system calls by the isolation monitor **116** and executed through basic computation services **206** within the kernel. Other services, such as secured access to the video display and other user I/O devices including keyboard and mouse, are executed in a display and user I/O client **204** accessed using network protocols transported through communication pipes connected by the isolation monitor **116**. This is described in greater detail infra.

[0033] The isolation monitor **116** defines new OS APIs that include just the basic computation services, thus separating the basic primitives from rich application APIs. Then, the code that provides the rich application APIs is copied out of the operating system, from the non-isolated OS subsystems **112** (of FIG. 1), and into the application process (or sandbox environment) to the isolated OS subsystems **110**. For example, in a Windows implementation, the entire Win32

subsystem (e.g., Win32, COM, shell, printing, etc.) is copied into the application sandbox so that each application runs its own copy of the Win32 subsystem. In a Linux (or Apple™ operating system) implementation, for example, the subsystem can include X-Windows, display postscript, printing (e.g., common Unix printing system) and audio (e.g., advanced Linux sound architecture). (Although described in great detail with respect to Windows, as indicated above the disclosed architecture applies to other operating systems implementations as well.)

[0034] Since the Win32 subsystem now provides services to only a single application, the subsystem need not be protected with security checks or other mechanisms, such as placing it in another operating system process, from the application. Instead, the Win32 subsystem can be run in the same process as the application. The minimal computation interface required for the sandboxed environment is shown below.

[0035] The technique uses a remote user I/O server (e.g., server **122**) within the application appliance to provide a device driver interface to the Win32 subsystem, but then communicates (through a local communication channel) to the user interface services on the host OS via the display and user I/O client **204**. Application compatibility is preserved by reproducing the functionality that Windows provides in the operating system **108** (primarily from the non-isolated OS subsystems **112**) as components in the user-mode (as the isolated OS subsystems **110**), in the isolated process.

[0036] Continuing with the context of a Windows operating system, these components (the isolated application libraries **118** which provide OS API components of FIG. 1 and FIG. 2) can include, but are not limited to, the Win OS API module (e.g., which includes kernel32.dll, user32.dll, gdi32.dll), the “New Technology” NT API module (e.g., ntdll.dll, RDP display interface (e.g., rdpvdd.dll), and an interface to the isolation monitor **116** for services and other processes outside the isolated environment. Note that the equivalent services mentioned above are a subset of the services that can be employed in the OS subsystem.

[0037] The disclosed architecture utilizes an isolation-optimized interface by providing at least virtual memory management, file access, thread creation, pipes, system time, and cryptographically strong random bits. These basic computation services are a sufficient kernel substrate upon which to implement higher-level process services as libraries, such as a registry for configuration management, thread worker factories, and more sophisticated thread synchronization operations. Isolation is enforced by a combination of virtual memory hardware and a highly restricted kernel API exposed by the isolation monitor **116**. Communication is allowed only through pipes. Pipes may not be configured at runtime; instead, the pipes are declared in the application manifest that specifies the requisite files and pipes to other applications or system services (such as the desktop display).

[0038] The architecture application binary interface (ABI) exports the following abstractions (and each minimizes the OS state stored on behalf of the application, facilitating user-space process migration and future proofing).

[0039] File handles. Memory-mapped files are provided by which applications map in read-only text and data sections. Processes do not communicate through the file system. Following the principle of minimal OS state, the file handles have no cursor; sequential read() operations are managed by emulation in an isolation application library instead. Conceptually, file mapping can be implemented with a single map

system call. Since Windows programs first open a file and then map it, file handles are provided to connect open to map without breaking error handling code in applications.

[0040] Pipes. Inter-process communication (IPC) and blocking synchronization are accomplished with ordered, reliable, message-based pipes, equivalent to PF UNIX-domain SOCK_DGRAM pipes. When multiple threads attempt to read the same pipe concurrently, each message is delivered to a single reader. A DkPipeSelect() call is provided that returns when data is available. This is similar to the Posix (portable OS interface for Unix) convention, in which select and poll return when data is available. Standard Windows pipes have the convention that WaitForMultipleObjects() returns after data has been read, possibly on multiple channels. The return-on-read semantics makes simulating many NT™ (new technology) kernel functions needlessly complicated; therefore, return-on-available semantics are provided. Applications specify pipes to other applications or to the user interface, in the application manifest.

[0041] Threads and processes. ABIs are provided for thread creation and process creation. Creating a process is more than just creating a thread in a new address space; the kernel also evaluates a new manifest and creates new pipe and file relationships. As part of process creation, the parent may request a pipe to the child. To maintain isolation, a process or thread may only terminate itself; there is no ABI to terminate, change the memory mapping of, or otherwise change the state of a separate process or thread.

[0042] GUI access. A feature for enabling a narrow isolation boundary is the use of a minimal pixel-blitting interface. Conventional GUI (graphical user interface) APIs such as in Windows and X11 expose a variety of abstractions, for example, windows, widgets, fonts, menus, events, callbacks, and much more. In contrast, the disclosed architecture moves all of the rendering and event loop functionality into the application itself, exposing only simple pixel-blit access to the trusted display, and a one-way flow of low-level keyboard and mouse input messages.

[0043] RDP background. The remote user I/O server **122** and the display and user I/O client **204** exchange messages using the remote desktop protocol (RDP), a protocol designed to achieve bandwidth-efficient remote display access. Its application-side component is a video driver and frame buffer that absorbs the output of the rich GUI framework above it. RDP harvests pixmaps (pixel maps) off the frame buffer and transmits the pixmaps to the display component. Mouse click and keystroke events from the display component are sent to the application component and injected into the GUI framework as if from local devices. RDP encapsulates the complexity of the GUI framework on one side of the channel, exposing only a conceptually simple pixel-blitting interface.

[0044] RDP exploits this interface simplicity to insert a variety of compression and coding techniques, and even profile-driven adaptive meta-protocols. This is a simple display-side code base, and a simple protocol amenable to sanitization. Essentially, RDP minus compression is a simple blit interface; the work of converting the GUI API to pixels on the application side and the work of blitting pixels on the display side has been done.

[0045] The previous application-side implementation of RDP is a kernel-mode display driver: it provides a frame buffer target for the output of the lowest layers of the Windows GUI stack, identifies changed pixels, and ships buckets of pixels to the display side. The architecture, in repackaging

the kernel-side layers of the Windows GUI stack as in-process application libraries, also links in the application-side components of RDP in the remote user I/O server **122**.

[0046] The display-side component, the user I/O client **204**, retains the task of asking the hardware abstracting components **114**, such as the display, to render the pixels received from the application-side implementation of RDP in the remote user I/O server **122**. The architecture uses the existing Windows-based RDP client implementation, stripped down to remove unneeded compression modules to maximize robustness.

[0047] A benefit of the blit-based approach, realized by the RDP protocol, is that it is stateless, isolated, and gracefully handles disconnection. This property is utilized to transparently decouple application logic from the user interface, which simplifies the task of process migration. Rather than serializing and migrating complex kernel data structures, these data structures travel in-place in the application's memory image, where the structures were created by the isolated OS subsystems **110**.

[0048] With respect to refactoring Windows, the architecture moves code out of the kernel or re-implements services in user-level libraries. The kernel portion of the Windows subsystem (win32k) is ported from kernel modules to a user-level dynamically-linked library. A portion of the NT kernel API is also re-implemented in a user library on top of the application subsystem kernel API.

[0049] Following is background about the Windows OS. In a Windows system, an application and its libraries occupy a process along with system-supplied user-mode libraries that provide interfaces to the core system services (ntdll, similar to the Unix libc) and to the graphical user interface (user32 and gdi32, the equivalent of Unix libX11 and higher-level libraries such as libgtk). The NT kernel implements the core of a monolithic operating system: resource management, scheduling, device drivers, file system, and the registry, a key-value store for configuration data. The Windows subsystem (win32k) provides the analogue of an X server, a print server (e.g., the Common Unix Printing System), and audio support (e.g., Advanced Linux Sound Architecture).

[0050] There are two system daemons in Windows: csrss and wininit. Csrss (the Client/Server Runtime SubSystem) is the first user mode process started during boot, the analogue of the Unix init daemon. Csrss' system initialization duties also include preloading kernel caches with public data to be shared among all processes, such as the default fonts, internationalization tables, and cursors. The wininit daemon launches the components of the user's desktop upon login, the analogue of gnome-session. Each new process contacts csrss, which establishes a shared-memory segment between the shared process and win32k used to save kernel-crossings for read-only GUI operations.

[0051] The disclosed architecture preserves application compatibility by reproducing the functionality Windows provides in the kernel as components of the user-mode, isolated process. The kernel GUI components, including both the general win32k library and the video driver implemented by the RDP server, are moved directly into the subsystem process (the former is part of the isolated OS subsystems **110** and the latter is the remote user I/O server **122**). The ntdll interface library is preserved, but rather than calling into the kernel, it now calls an NT.shim library, an implementation that simulates the kernel features expected by most applications (part of the isolated OS subsystems **110**).

[0052] The isolated process user interface is exposed to the real world via an RDP display client (the user I/O client 204) which accesses the Windows kernel through conventional APIs. In other words, the user I/O client 204 is a non-isolated application 106, which uses the non-isolated OS subsystems 112.

[0053] With respect to isolation, a well-isolated process is a useful mechanism. This is exploited by introducing policies in the form of the application firewall. Users specify simple, coarse rules that either protect sensitive data and applications (“allow only these two applications to touch this financial data”) or rules that confine untrusted applications (“disallow this downloaded game from touching any of my data”). A collection of such rules forms an application firewall. The rules map to approving or denying application manifest requests.

[0054] Applications specify requirements for external resources and communication pipes with the application manifest. The application manifest specifies which resources are required and which are optional; if an optional pipe is not available, the application loses non-critical functionality. An application’s manifest requests a set of IPC pipes. For each pipe, the manifest gives the external name of the pipe, an internal identifier, and a flag indicating which pipes can tolerate disconnection for migration.

[0055] Since all inter-process communication goes through declared pipes, an application firewall can impose information flow rules, ruling out particular pipes, or specifying ALLOW or DENY lists of applications that may connect to a given pipe endpoint. The application firewall can be configured by the user during application installation.

[0056] In one implementation, each application (e.g., isolated application 104) is distributed with all of its requisite files, including supporting libraries, fonts, and internationalization tables. In an alternative implementation, an application’s manifest may also specify access to “My Music” or “My Documents”, which the user’s firewall may approve or deny.

[0057] FIG. 3 illustrates future proofing in which the isolated application 104 which runs in a first secure application execution system 300 can also be run in a second secure application execution system 301. As previously described in FIG. 1, the first secure application execution system 300 includes the first operating system 108 with the isolation monitor 116 (denoted here as a first application monitor). The second secure application execution system 301 includes a second operating system 308 with a second isolation monitor 316. The basic computation services exposed by the second isolation monitor 316 are compatible with the basic computation services exposed by the first isolation monitor 116.

[0058] When run on the second operating system 308, the isolated application 104 is placed in a different isolation container 302 as is compatible and provided by the second OS 308, and isolated application 104 uses the exact same application code and the same code for the same isolated application libraries 118, isolated OS subsystems 110, and remote user I/O server 122. Providing compatibility between the first operating system 108 and the second operating system 308 is straightforward with the disclosed architecture, because the rich APIs that are often large in number and have complex semantics which are captured in the isolated OS subsystems 110. The isolated application 104 runs with the same rich

APIs in the isolated OS subsystems 110 whether it runs on the first operating system 108 or the second operating system 308.

[0059] Note that the isolation containers (102 and 302) can both be run on the same computer or each on a different computer. Note also that the operating systems (108 and 308) can be the same type (e.g., Win XP) of operating system each run on a different computer, the same single operating system (OS 108 is the same operating system as OS 308) running on a single computer, different type of operating systems (e.g., Win XP versus Win 7) running on the same computer (e.g., via virtual machines, multi-boot configuration, etc.), and so on.

[0060] For example, using the described architecture, a newer Windows operating system (e.g., Windows 7™) can be made to run applications written for the Windows XP operating systems when those applications are combined in an isolation container with Windows XP isolated OS subsystems, and the Windows 7 operating system runs an isolation monitor that exposes a set of basic computation services compatible with the isolation monitor targeted by the Windows XP isolated OS systems.

[0061] Conversely, using the described architecture, the Windows XP operating system can run applications written for the Windows 7 operating system when those applications are combined in an isolation container with Windows 7 isolated OS subsystems and the Windows XP operating system runs an isolation monitor that exposes a set of basic computation services compatible with the isolation monitor targeted by the Windows 7 isolated OS subsystems.

[0062] In yet another implementation, an application (e.g., isolated application 104) that normally runs on a Vendor A operating system (OS 108) can be made to run on a Vendor B operating system (OS 308, which is different than the Vendor A operating system) by configuring an isolation monitor (the isolation monitor 316) of the Vendor B operating system to interface to the Vendor B operating system, and also interface to the isolated OS subsystem (isolated subsystem 110) that facilitates running of the application on the Vendor B operating system.

[0063] In a more specific example of the above generalization using Windows and Apple programs (but also applies to any mix of programs and operating systems), the isolated application 104 of the secure application execution system 300 (e.g., Windows application running on a Windows operating system) is now desired to be run in the second secure application execution system 301 of an Apple operating system (a Windows application on an Apple operating system).

[0064] To make this work, the second isolation monitor 316 is designed to interface to the Apple OS (the second OS 308) and expose a set of basic computation services compatible with the Windows-based isolated OS subsystem 110 (as used in the first isolation container 102, but now also used in the second isolation container 302). Those skilled in the art will recognize that creating a compatible isolation monitor is relatively straightforward because of the small number and simple semantics of the basic computation services (e.g., in one implementation, the isolation monitor is fewer than 5,000 lines of C++ code). This is in contrast with the large number and complex semantics of the rich APIs in the isolated OS subsystems (e.g., one implementation of the Windows Win32 subsystem is over one million lines of C and C++ code).

[0065] Put another way, a secure application execution system is provided that comprises an isolation container in which

an application for a first OS runs in isolation, the isolation container formed in association with a second OS, an isolated OS subsystem that runs in the isolation container in association with and interfaces to the application to provide rich functionality to the application, and an isolation monitor of the second OS that interfaces basic computation services of the second OS to the isolated OS subsystem to enable the application to run in isolation on the second OS. The basic computation services include at least one of virtual memory management, thread creation, or thread synchronization. The isolated application uses a corresponding remote user I/O server to communicate with a user I/O client outside the isolation container.

[0066] The rich functionality provided by the isolated OS subsystem includes at least one of a graphical user interface service, an application configuration management service, a printer service, or an audio service. The isolated application uses a corresponding remote user I/O server to communicate with a user I/O client outside the isolation container. The isolated application is migrated to a second computing environment by reading from some or all of an address space of the isolation container, which is in a first computing environment. The isolation monitor employs a collection of rules that map from an application manifest to approval or denial of resource requests, the manifest defines which resources outside the isolation container are available to the isolated application.

[0067] FIG. 4 illustrates future proofing system 400 in which the operating system 108 and the isolation monitor 116 can run the first isolated application 104 with the first set of isolated OS subsystems 110 and can run a second isolated application 404 with a second set of isolated OS subsystems 410, and both isolated OS subsystems (110 and 410) use basic computation services exposed through the same isolation monitor 116. The set of rich APIs exposed by the first set of isolated OS subsystems 110 differs in number or semantics from the second set of isolated OS subsystems 410. The second isolated application 404 is run in a second isolation container 402 that includes the second isolated application 404, a set of second isolated application libraries 418, the second set of isolated OS subsystems 410, and a second remote user I/O server 422.

[0068] If the second set of OS subsystems 410 provides sufficient compatibility with the first set of OS subsystems 110, the second remote user I/O server 422 may be the same as the first remote user I/O server 122. Likewise, the second isolated application libraries 418 may be the same as the first isolated application libraries 118. Still further, the second isolated application 404 may be the same as the first isolated application 104.

[0069] For example, a Windows 7 operating system can be made to run applications written for the Windows XP, Windows Vista, or Windows 7 operating systems when those applications are combined in associated isolation containers with Windows XP, Windows Vista™, or Windows 7 isolated OS subsystems, respectively, and the Windows 7 operating system runs an isolation monitor compatible with the isolation monitors targeted by the Windows XP isolated OS subsystems, the Windows Vista isolated OS subsystems, or the Windows 7 isolated OS subsystems. Those skilled in the art will recognize that the modifications made to make a first set of isolated OS subsystems, such as the Windows 7 isolated OS subsystems, run on an isolation monitor can be reused to make a second set of isolated OS subsystems, such as the Windows

XP isolated OS subsystems, run on the same isolation monitor. This is the case because the basic computation services provided by an isolation monitor are not tailored to a specific isolated OS subsystem, but instead provide simple semantics general to many isolated OS subsystems.

[0070] Put another way, a secure application execution system is provided that comprises a first isolation container in which a first isolated application runs in isolation, and a second isolation container in which a second isolated application runs in isolation, the first isolated application and the second isolated application running in association with a single OS. The system further includes a first isolated OS subsystem of the first isolation container that provides services to the first isolated application, a second isolated OS subsystem of the second isolation container that provides services to the second isolated application, and an isolation monitor via which basic computation services are provided to each of the first isolated OS subsystem and the second isolated OS subsystem. The basic computation services include virtual memory management, threads creation, and thread synchronization.

[0071] The rich functionality includes at least one of the isolated OS subsystems, the isolated OS subsystems comprise at least one of a graphical user interface service, an application configuration management service, a printer service, or an audio service. At least one of the first isolated application or the second isolated application uses a corresponding remote user I/O server to communicate with a user I/O client outside of a corresponding isolation container. The first isolated application uses a first corresponding remote user I/O server and the second isolated application uses a second corresponding remote user I/O server, and the first corresponding remote user I/O server and the second corresponding remote user I/O server both communicate with a first user I/O client outside the isolation containers.

[0072] In yet another implementation, a secure application execution system is provided that comprises an isolated OS subsystem that runs in an isolation container and provides services to an isolated application equivalent to services provided by a non-isolated OS subsystem to a non-isolated application. The isolated OS subsystem receives basic computation services from an isolation monitor in an OS that provides similar basic computation services to the non-isolated OS subsystem. The basic computation services received include virtual memory management, thread creation, and thread synchronization. The equivalent services include at least one of GUI services, application configuration management services, printer services, or audio services.

[0073] With respect to process migration, the disclosed architecture uses a pipe disconnectable flag in the manifest to assess whether a process can be migrated. If every pipe from a process is either disconnectable, or the process on the other end can migrate along with the process, then the process may be migrated. By bundling the state and complexity of the GUI into the process itself, a large class of dependencies on the kernel that typically could make migration difficult, are eliminated and replaced with RDP's reconnectable protocol. Disruption by reconnections is tolerated, since many pipes will be to Internet services.

[0074] A challenge is plumbing isolated processes to the reference monitor, adapting the NT APIs, repackaging the win32k GUI library, replacing the registry, repackaging COM, and organizing the implementation to facilitate easy migration.

[0075] The architecture basic computation API is implemented inside of the isolation monitor **116** (called Dkmon in one implementation).

[0076] When Dkmon starts a new process, it creates a suspended Windows process, specifying the dkinit application loader as the binary. The Windows kernel then creates an address space, maps in dkinit and the system-wide ntdll library, and suspends execution at ntdll's entry point. ntdll is the analog of the Unix/lib/ld.so, but in Windows, the kernel installs a particular version of ntdll at the same virtual address in every process, and makes upcalls to functions at fixed offsets into the library. ntdll is modified to make calls. To that end, Dkmon maps DkNtdll into the new process' virtual memory, then patches the system-provided ntdll, overwriting its functions with jumps to DkNtdll; the system library is eviscerated to a jump table.

[0077] Dkmon writes a parameter block into the process, communicating initialization parameters such as the paths of the manifest and checkpoint file.

[0078] Dkmon resumes the suspended process, causing DkNtdll to set up initial library linkage, including the win32k library, and transfer control to dkinit. Dkinit invokes the loader (DkNtdll) dynamically to load the application and its imported libraries, and jumps to the application's entry point.

[0079] To avoid Time-Of-Check-To-Time-Of-Use concurrency vulnerabilities, Dkmon copies in system call arguments exactly once. By reducing the shared application state in the kernel, as well as enforcing coarse isolation policies, exposure to state inconsistency is minimized.

[0080] In order to provide binary compatibility with existing desktop applications, user space implementations of many NT kernel functions are provided in the isolated OS subsystems **110**. In some cases, such as allocating virtual memory or opening a file, the NT function is a thin layer that calls the isolation monitor **116**. In other cases, such as the synchronization mechanisms, the implementation can be more involved.

[0081] The NT kernel API exposes several blocking synchronization primitives with varying semantics, including events, mutants (mutexes), and semaphores. Basic features of these synchronization primitives can be implemented with non-blocking locks and user-level data structures. Functionally, synchronization in the user space using blocking semantics is facilitated by providing a wait queue inside the kernel when the user space lock is contended. The signaling mechanism is a pipe. When a process blocks on a synchronization handle, such as a mutant, the process blocks waiting for data to become available in a pipe associated with the event. When a process releases a mutant, the process writes a byte to the pipe and a blocked process is awakened and reads the byte. Only one process will read the byte, so only one process will acquire the mutant.

[0082] Several applications wait on one or more timer handles. Dkmon supplies only DkSystemTimeQuery and the ability to block on time via a timeout argument to DkPipeSelect. The application shim library uses DkSystemTimeQuery to normalize relative timeouts to absolute timeouts. The shim provides timer multiplexing by DkPipeSelecting on the earliest timeout among the application-specified handles.

[0083] A challenge in porting win from a kernel library to a user space DLL (dynamic linked library) is to reproduce its complicated, multi-process initialization sequence. First, the single, system-wide instance of the win32k module is initialized in kernel space. Second, a csrss-spawned user space

process preloads win32k's caches with shared public objects such as fonts and bitmaps. To fetch an object into its cache, win32k makes upcalls to user32 and gdi32 DLLs, so the user-space process first loads those dlls before filling the cache. Third, when an ordinary user process starts, the process loads its own copies of user32 and gdi32, which connect to win32k and provide GUI service.

[0084] The architecture bootstrap first loads and initializes its copy of win32k, then loads user32 and gdi32 without calling the respective initializers, and then fills the win32k caches. Now win32k is completely initialized, so the bootstrap calls user32's and gdi32's real library initialization functions. Each DLL has been loaded by the standard loader, so at this point, the bootstrap can request the loader to load the user program, and the program's dependencies on user32 and gdi32 will be satisfied with the extant instances now bound to win32k.

[0085] The read-only shared-memory segment established by csrss is now established as a shared heap, since the two components that access it, win32k and user 32, share a protection domain. Synchronization code and shim code is provided to get win32k running in the user space.

[0086] Windows' kernel object manager manages a hierarchical namespace, mapping paths to drivers that implement the named objects (analogous to the vnodes that tie files, devices, and /proc together in Unix). The Windows registry is an object manager instance that provides a hierarchical key-value store. The disclosed architecture refactors the OS relationship to make applications self-contained. Thus, the NT shim supplies a registry implementation with no transactions and coarse locking. Each application has a private registry image generated by running the application in Windows. The instrumentation records the set of opened keys, snapshots the values in the Windows registry, and emits a registry image.

[0087] Refactoring the COM (component object model) subsystem follows the same basic pattern: application-side libraries expect to communicate with a separate server process. An instance of the server code is linked as isolated OS subsystems **110** library inside the process, and a thread is created to run it. The application-side library is linked directly to the server, cutting out the RPC (remote procedure call) stubs.

[0088] Migration can be implemented entirely in user space by tracking the layout of the address space, threads, and handles in user space. To checkpoint an application, the contents of the address space (including this bookkeeping) are written to a file. In order to initiate a checkpoint, the reference monitor writes a bit into the loader block. Each thread checks this bit before issuing a system call and periodically while waiting on input from a pipe. Each thread then checkpoints its register state and terminates without deleting its stack. The last thread to exit actually performs the copy of the address space into the file.

[0089] In order to resume from a checkpoint, the application performs basic loader initialization steps, then loads the checkpoint file. The resuming application then restores all anonymous (non-file backed) memory, followed by the private handles, and finally restores file mappings. Externally visible handles are loaded by the manifest as usual. The application then recreates the threads, forming thread execution blocks (TEB) to ensure thread identifiers match those in the checkpointed image. By moving process abstractions into the process itself, the architecture makes the migration task straightforward.

[0090] Again, with respect to inter-process communications, the application manifest specifies whether a channel can be broken; processes with unbreakable connections are migrated together. The disclosed architecture makes connections to hardware resources, such as the window manager, stateless and thereby supports disconnection and reconnection without loss of function, and allows independent migration of application logic and the graphical user interface.

[0091] In addition to migrating a process' address space and IPC connections, state stored inside the operating system is also migrated. The disclosed architecture migrates processes across disjoint operating systems with matching ABIs. This is made possible by making all inter-process communication channels explicit and minimizing OS state that needs to be tracked and restored, thereby enabling the migration of processes entirely at user-level.

[0092] A minimal exemplary computation interface utilized for the sandboxed environment is described as follows.

```
// Virtual Memory
DKSTATUS
DkVirtualMemoryAllocate(
    __inout PVOID *BaseAddress,
    __inout PSIZE_T RegionSize,
    __in ULONG AllocationType,
    __in ULONG Protect);
DKSTATUS
DkVirtualMemoryFree(
    __in PVOID BaseAddress,
    __in SIZE_T RegionSize,
    __in ULONG FreeType);
DKSTATUS
DkVirtualMemoryProtect(
    __inout PVOID BaseAddress,
    __inout SIZE_T RegionSize,
    __in ULONG NewProtect,
    __out PULONG OldProtect);
// IPC
BOOL
DkPipeFork(
    __in HANDLE Handle,
    __out PULONG64 Token,
    __out PHANDLE NewHandle);
BOOL
DkSelfPipeCreate(
    __out PHANDLE Handle1,
    __out PHANDLE Handle2,
    __out PULONG64 Token);
ULONG
DkPipeRead(
    __in HANDLE Handle,
    __in BOOL Async,
    __in PVOID AsyncToken,
    __inout PVOID *Buffer,
    __in ULONG Length,
    __in_opt PLONG64 Timeout);
ULONG
DkPipeWrite(
    __in HANDLE Handle,
    __in BOOL Async,
    __in PVOID AsyncToken,
    __in PVOID Buffer,
    __in ULONG Length);
ULONG
DkPipeSelect(
    __in ULONG Count,
    __in const HANDLE *Handles,
    __in_opt PLONG64 Timeout);
ULONG
DkPipePeek(
    __in HANDLE Handle);
// Isolated File Access
```

-continued

```
PVOID
DkFileOpen(
    __in PUNICODE_STRING pUri,
    __in_opt PVOID DesiredAddress,
    __in ACCESS_MASK DesiredAccess,
    __in ULONG ShareMode,
    __in ULONG CreateDisposition,
    __in ULONG CreateOptions,
    __in SIZE_T Offset,
    __inout_opt PSIZE_T ViewSize);
BOOL
DkFileTruncate(
    __in PUNICODE_STRING Uri,
    __in SIZE_T Length);
DKSTATUS
DkFileUnmap(
    __in PVOID addr);
BOOL
DkFileSync(
    __in PVOID addr);
BOOL
DkFileUnlink(
    __in PUNICODE_STRING Uri);
DKSTATUS
DkFileAttributesQuery(
    __in PUNICODE_STRING Uri,
    __out PDK_FILE_ATTRIBUTES Attrs);
// Threading
BOOL
DkThreadCreate(
    __in SIZE_T StackSize,
    __in PDK_THREAD_START Address,
    __in_opt PVOID Parameter,
    __in ULONG CreationFlags,
    __out_opt PHANDLE Pipe,
    __out_opt PULONG64 PipeToken);
VOID
DkThreadExit( );
BOOL
DkProcessCreate(
    __in_opt PUNICODE_STRING Appl,
    __in_opt PUNICODE_STRING CmdLin,
    __out_opt PHANDLE Pipe,
    __out_opt PULONG64 PipeToken);
VOID
DkProcessExit( );
// Other
BOOL
DkSystemTimeQuery(
    __out PLONG64 SystemTime);
BOOL
DkRandomBitsRead(
    __in out PVOID Buf,
    __in SIZE_T BufSize);
BOOL
DkDebugOutput(
    __in PUNICODE_STRING Message);
```

[0093] Included herein is a set of flow charts representative of exemplary methodologies for performing novel aspects of the disclosed architecture. While, for purposes of simplicity of explanation, the one or more methodologies shown herein, for example, in the form of a flow chart or flow diagram, are shown and described as a series of acts, it is to be understood and appreciated that the methodologies are not limited by the order of acts, as some acts may, in accordance therewith, occur in a different order and/or concurrently with other acts from that shown and described herein. For example, those skilled in the art will understand and appreciate that a methodology could alternatively be represented as a series of inter-related states or events, such as in a state diagram. Moreover, not all acts illustrated in a methodology may be required for a novel implementation.

[0094] FIG. 5 illustrates a method of creating secure application execution in accordance with the disclosed architecture. At 500, in an operating system kernel, identify rich (non-minimal) functionality from minimal requisite functionality (the basic computation services 206) associated with running an application. The minimal requisite functionality is identified and exposed to the applications through the isolation monitor. At 502, the rich functionality is moved from the kernel into user-space libraries (e.g., in the isolated OS subsystems). At 504, communications between the rich functionality and the kernel is implemented via an isolation monitor. At 506, the rich functionality is isolated from the kernel using an application firewall of rules that control interaction between the functionality and the kernel (and other components such as the user I/O client).

[0095] FIG. 6 illustrates further aspects of the method of FIG. 5 for converting additional non-isolated OS subsystems either into isolated OS subsystems or into external network service, such as the user I/O client. Note that the arrowing indicates that each block represents a step that can be included, separately or in combination with other blocks, as additional aspects of the method represented by the flow chart of FIG. 5. At 600, the rich functionality is run in an application process of the application or external to the application process. At 602, a network interface is provided between the application and the functionality for communicating with host operating system services via a server. At 604, optional external resources, requisite external resources, and communications pipes, to other applications and system services, are specified in an application manifest. At 606, an interface to host operating system services is via a kernel interface implemented inside the isolation monitor. At 608, rich functionality services are moved from the operating system kernel into an isolated OS subsystem, which services include windowing, access control, and user interfaces.

[0096] FIG. 7 illustrates a method of factoring operating system code into components to be used in an application appliance environment. At 700, a system component that exists outside the application process and which provides a required service for the application process is identified. At 702, a check is made if the resources managed by the component need to be shared. If so, flow is to 704 where a network protocol is chosen to be used to access the shared resource. Flow is then to 706, where the application appliance is augmented with code that implements the network protocol. At 708, the system component is then accessed as a network service.

[0097] If, at 702, the resources exposed by the system component do not need to be shared with other applications, then, flow is to 710, where the code is copied into the application appliance. For example, the physical keyboard, mouse, and video display are shared devices; thus, in one embodiment, the remote desktop protocol (RDP) can be employed to access the shared display at 704 and add RDP server support to the remote user I/O server in step 706 before modifying the Win32k part of the isolated OS subsystems to use the RDP server code introduced in step 704.

[0098] At 712, as the component is copied into the application appliance, any code that requests security authentication can be removed, disabled, or modified to grant access. At 714, as the component is copied into the application appliance, any code that provides enforcement of security isolation policies can be removed or disabled. The code can be removed or disabled (or modified to grant access), because

the code is now inside the application appliance, and therefore, will not protect any other services from an errant or malicious application appliance.

[0099] As used in this application, the terms “component” and “system” are intended to refer to a computer-related entity, either hardware, a combination of software and tangible hardware, software, or software in execution. For example, a component can be, but is not limited to, tangible components such as a processor, chip memory, mass storage devices (e.g., optical drives, solid state drives, and/or magnetic storage media drives), and computers, and software components such as a process running on a processor, an object, an executable, a module, a thread of execution, and/or a program. By way of illustration, both an application running on a server and the server can be a component. One or more components can reside within a process and/or thread of execution, and a component can be localized on one computer and/or distributed between two or more computers. The word “exemplary” may be used herein to mean serving as an example, instance, or illustration. Any aspect or design described herein as “exemplary” is not necessarily to be construed as preferred or advantageous over other aspects or designs.

[0100] Referring now to FIG. 8, there is illustrated a block diagram of a computing system 800 that executes application sandboxing in accordance with the disclosed architecture. In order to provide additional context for various aspects thereof, FIG. 8 and the following description are intended to provide a brief, general description of the suitable computing system 800 in which the various aspects can be implemented. While the description above is in the general context of computer-executable instructions that can run on one or more computers, those skilled in the art will recognize that a novel embodiment also can be implemented in combination with other program modules and/or as a combination of hardware and software.

[0101] The computing system 800 for implementing various aspects includes the computer 802 having processing unit(s) 804, a computer-readable storage such as a system memory 806, and a system bus 808. The processing unit(s) 804 can be any of various commercially available processors such as single-processor, multi-processor, single-core units and multi-core units. Moreover, those skilled in the art will appreciate that the novel methods can be practiced with other computer system configurations, including minicomputers, mainframe computers, as well as personal computers (e.g., desktop, laptop, etc.), hand-held computing devices, micro-processor-based or programmable consumer electronics, and the like, each of which can be operatively coupled to one or more associated devices.

[0102] The system memory 806 can include computer-readable storage (physical storage media) such as a volatile (VOL.) memory 810 (e.g., random access memory (RAM)) and non-volatile memory (NON-VOL.) 812 (e.g., ROM, EPROM, EEPROM, etc.). A basic input/output system (BIOS) can be stored in the non-volatile memory 812, and includes the basic routines that facilitate the communication of data and signals between components within the computer 802, such as during startup. The volatile memory 810 can also include a high-speed RAM such as static RAM for caching data.

[0103] The system bus 808 provides an interface for system components including, but not limited to, the system memory 806 to the processing unit(s) 804. The system bus 808 can be

any of several types of bus structure that can further interconnect to a memory bus (with or without a memory controller), and a peripheral bus (e.g., PCI, PCIe, AGP, LPC, etc.), using any of a variety of commercially available bus architectures.

[0104] The computer **802** further includes machine readable storage subsystem(s) **814** and storage interface(s) **816** for interfacing the storage subsystem(s) **814** to the system bus **808** and other desired computer components. The storage subsystem(s) **814** (physical storage media) can include one or more of a hard disk drive (HDD), a magnetic floppy disk drive (FDD), and/or optical disk storage drive (e.g., a CD-ROM drive DVD drive), for example. The storage interface(s) **816** can include interface technologies such as EIDE, ATA, SATA, and IEEE 1394, for example.

[0105] One or more programs and data can be stored in the memory subsystem **806**, a machine readable and removable memory subsystem **818** (e.g., flash drive form factor technology), and/or the storage subsystem(s) **814** (e.g., optical, magnetic, solid state), including an operating system **820** (e.g., OS **108** and OS **308**), one or more application programs **822** (e.g., isolated application **104**, non-isolated application **106**, and isolated application **404**), other program modules **824** (e.g., isolated application libraries **118** and non-isolated application libraries **120**), and program data **826**.

[0106] The one or more application programs **822**, other program modules **824**, and program data **826** can include the entities and components of the system **100** of FIG. 1, entities and components of the system **200** of FIG. 2, the entities and components of FIG. 3, the entities and components of the system **400** of FIG. 4, and the methods represented by the flowcharts of FIGS. 5-7, for example.

[0107] Generally, programs include routines, methods, data structures, other software components, etc., that perform particular tasks or implement particular abstract data types. All or portions of the operating system **820**, applications **822**, modules **824**, and/or data **826** can also be cached in memory such as the volatile memory **810**, for example. It is to be appreciated that the disclosed architecture can be implemented with various commercially available operating systems or combinations of operating systems (e.g., as virtual machines).

[0108] The storage subsystem(s) **814** and memory subsystems (**806** and **818**) serve as computer readable media for volatile and non-volatile storage of data, data structures, computer-executable instructions, and so forth. Such instructions, when executed by a computer or other machine, can cause the computer or other machine to perform one or more acts of a method. The instructions to perform the acts can be stored on one medium, or could be stored across multiple media, so that the instructions appear collectively on the one or more computer-readable storage media, regardless of whether all of the instructions are on the same media.

[0109] Computer readable media can be any available media that can be accessed by the computer **802** and includes volatile and non-volatile internal and/or external media that is removable or non-removable. For the computer **802**, the media accommodate the storage of data in any suitable digital format. It should be appreciated by those skilled in the art that other types of computer readable media can be employed such as zip drives, magnetic tape, flash memory cards, flash drives, cartridges, and the like, for storing computer executable instructions for performing the novel methods of the disclosed architecture.

[0110] A user can interact with the computer **802**, programs, and data using external user input devices **828** such as a keyboard and a mouse. Other external user input devices **828** can include a microphone, an IR (infrared) remote control, a joystick, a game pad, camera recognition systems, a stylus pen, touch screen, gesture systems (e.g., eye movement, head movement, etc.), and/or the like. The user can interact with the computer **802**, programs, and data using onboard user input devices **830** such as a touchpad, microphone, keyboard, etc., where the computer **802** is a portable computer, for example. These and other input devices are connected to the processing unit(s) **804** through input/output (I/O) device interface(s) **832** via the system bus **808**, but can be connected by other interfaces such as a parallel port, IEEE 1394 serial port, a game port, a USB port, an IR interface, etc. The I/O device interface(s) **832** also facilitate the use of output peripherals **834** such as printers, audio devices, camera devices, and so on, such as a sound card and/or onboard audio processing capability.

[0111] One or more graphics interface(s) **836** (also commonly referred to as a graphics processing unit (GPU)) provide graphics and video signals between the computer **802** and external display(s) **838** (e.g., LCD, plasma) and/or onboard displays **840** (e.g., for portable computer). The graphics interface(s) **836** can also be manufactured as part of the computer system board.

[0112] The computer **802** can operate in a networked environment (e.g., IP-based) using logical connections via a wired/wireless communications subsystem **842** to one or more networks and/or other computers. The other computers can include workstations, servers, routers, personal computers, microprocessor-based entertainment appliances, peer devices or other common network nodes, and typically include many or all of the elements described relative to the computer **802**. The logical connections can include wired/wireless connectivity to a local area network (LAN), a wide area network (WAN), hotspot, and so on. LAN and WAN networking environments are commonplace in offices and companies and facilitate enterprise-wide computer networks, such as intranets, all of which may connect to a global communications network such as the Internet.

[0113] When used in a networking environment the computer **802** connects to the network via a wired/wireless communication subsystem **842** (e.g., a network interface adapter, onboard transceiver subsystem, etc.) to communicate with wired/wireless networks, wired/wireless printers, wired/wireless input devices **844**, and so on. The computer **802** can include a modem or other means for establishing communications over the network. In a networked environment, programs and data relative to the computer **802** can be stored in the remote memory/storage device, as is associated with a distributed system. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers can be used.

[0114] The computer **802** is operable to communicate with wired/wireless devices or entities using the radio technologies such as the IEEE 802.xx family of standards, such as wireless devices operatively disposed in wireless communication (e.g., IEEE 802.11 over-the-air modulation techniques) with, for example, a printer, scanner, desktop and/or portable computer, personal digital assistant (PDA), communications satellite, any piece of equipment or location associated with a wireless detectable tag (e.g., a kiosk, news

stand, restroom), and telephone. This includes at least Wi-Fi (or Wireless Fidelity) for hotspots, WiMax, and Bluetooth™ wireless technologies. Thus, the communications can be a predefined structure as with a conventional network or simply an ad hoc communication between at least two devices. Wi-Fi networks use radio technologies called IEEE 802.11x (a, b, g, etc.) to provide secure, reliable, fast wireless connectivity. A Wi-Fi network can be used to connect computers to each other, to the Internet, and to wire networks (which use IEEE 802.3-related media and functions).

[0115] What has been described above includes examples of the disclosed architecture. It is, of course, not possible to describe every conceivable combination of components and/or methodologies, but one of ordinary skill in the art may recognize that many further combinations and permutations are possible. Accordingly, the novel architecture is intended to embrace all such alterations, modifications and variations that fall within the spirit and scope of the appended claims. Furthermore, to the extent that the term “includes” is used in either the detailed description or the claims, such term is intended to be inclusive in a manner similar to the term “comprising” as “comprising” is interpreted when employed as a transitional word in a claim.

What is claimed is:

1. A computer-implemented secure application execution system having computer readable media that store executable instructions executed by a processor, comprising:

an isolation container in which an isolated application runs in isolation from a non-isolated application, the isolated application and non-isolated application both running in association with a single operating system (OS); and
an isolated OS subsystem of the isolation container that provides services to the isolated application, and a non-isolated OS subsystem of the OS that provides services to the non-isolated application, the isolated OS subsystem and non-isolated OS subsystem providing equivalent services to the corresponding isolated application and non-isolated application.

2. The system of claim **1**, wherein the isolated application and the non-isolated application use basic computation services provided by the OS.

3. The system of claim **2**, wherein the isolated application receives access to the basic computation services through an isolation monitor.

4. The system of claim **3**, wherein the isolated OS subsystem receives basic computation services through the isolation monitor.

5. The system of claim **3**, wherein the isolation monitor employs a collection of rules that map from an application manifest to approval or denial of resource requests, the manifest defines which resources outside the isolation container are available to the isolated application.

6. The system of claim **2**, wherein the basic computation services include at least one of virtual memory management, thread creation, or thread synchronization.

7. The system of claim **1**, wherein the equivalent services include at least one of a graphical user interface (GUI) service, an application configuration management service, a printer service, or an audio service.

8. The system of claim **1**, wherein the isolated application uses a remote user input/output (I/O) server that communicates with a user I/O client outside the isolation container.

9. The system of claim **1**, wherein the isolated application is migrated to a second computing environment by reading

from some or all of an address space of the isolation container, which isolation container is in a first computing environment.

10. A computer-implemented secure application execution system having computer readable media that store executable instructions executed by a processor, comprising:

an isolation container in which an application for a first OS runs in isolation, the isolation container formed in association with a second OS;

an isolated OS subsystem that runs in the isolation container in association with and interfaces to the application to provide rich functionality to the application; and

an isolation monitor of the second OS that interfaces basic computation services of the second OS to the isolated OS subsystem to enable the application to run in isolation on the second OS.

11. The system of claim **10**, wherein the basic computation services include at least one of virtual memory management, thread creation, or thread synchronization.

12. The system of claim **10**, wherein the rich functionality provided by the isolated OS subsystem includes at least one of a graphical user interface (GUI) service, an application configuration management service, a printer service, or an audio service.

13. The system of claim **10**, wherein the isolated application uses a corresponding remote user I/O server to communicate with a user I/O client outside the isolation container.

14. The system of claim **10**, wherein the isolated application is migrated to a second computing environment by reading from some or all of an address space of the isolation container, which is in a first computing environment.

15. The system of claim **10**, wherein the isolation monitor employs a collection of rules that map from an application manifest to approval or denial of resource requests, the manifest defines which resources outside the isolation container are available to the isolated application.

16. A computer-implemented secure application execution system having computer readable media that store executable instructions executed by a processor, comprising:

a first isolation container in which a first isolated application runs in isolation, and a second isolation container in which a second isolated application runs in isolation, the first isolated application and the second isolated application running in association with a single OS;

a first isolated OS subsystem of the first isolation container that provides services to the first isolated application, a second isolated OS subsystem of the second isolation container that provides services to the second isolated application; and

an isolation monitor via which basic computation services are provided to each of the first isolated OS subsystem and the second isolated OS subsystem.

17. The system of claim **16**, wherein the basic computation services include at least one of virtual memory management, threads creation, or thread synchronization.

18. The system of claim **16**, wherein the rich functionality includes at least one of the isolated OS subsystems, the isolated OS subsystems comprise at least one of a graphical user interface (GUI) service, an application configuration management service, a printer service, or an audio service.

19. The system of claim **16**, wherein at least one of the first isolated application or the second isolated application uses a

corresponding remote user I/O server to communicate with a user I/O client outside of a corresponding isolation container.

20. The system of claim 16, where the first isolated application uses a first corresponding remote user I/O server and the second isolated application uses a second corresponding

remote user I/O server, and the first corresponding remote user I/O server and the second corresponding remote user I/O server both communicate with a first user I/O client outside the isolation containers.

* * * * *