Virapix
A Portable Desktop Experience on Linux

By: Mandar Kashelikar

Abstract

It's not uncommon to wish for portable computing sans the prospect of trudging along with a laptop. Palmtops, PDAs and other smart gadgets are convenient alternatives, but what if you don't want to part with the power and the big screen of a PC? The solution is not a new kind of hardware but software that travels with you and that offers the same desktop experience wherever you go. *Virapix* is designed with this in mind.

Introduction

Virapix is a Linux based application that runs off a pen drive and hosts a self-contained desktop environment. It acts like a virtual machine to programs launched from within this environment and redirects all write operations and configuration changes to the USB drive. By virtualising the underlying operating system, Virapix isolates the host computer from any modifications that the user makes. By the same token, by confining all changes to the USB device, it empowers the user with the ability to carry the desktop in their pockets.

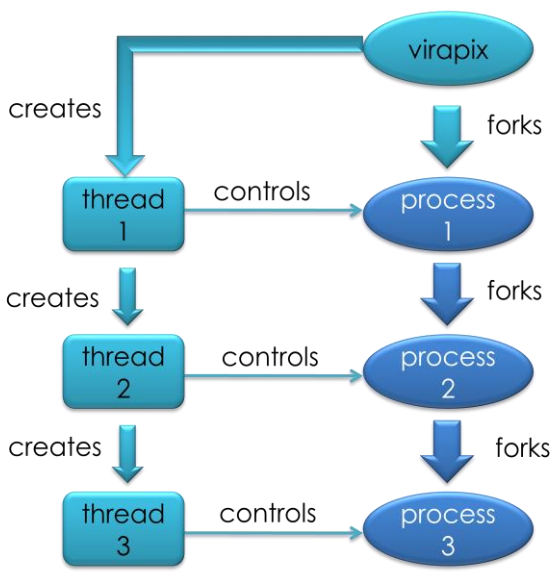
The Architecture

In this section, we'll have a detailed look at the architecture of *Virapix*. Strictly speaking, *Virapix* refers to a single executable that sits between Linux and the desktop software that it hosts. However the packaged software is a suite of applications that also includes *Fluxbox Window Manager, Mozilla Firefox, Open Office* and *Gvim*. The subject of this paper being *Virapix* alone, we'll exclude all the third party applications from the discussion.

The Top Level View

Let's begin with a simple depiction of process flow. As figure (1) shows, *Virapix* main process launches the parent application (the window manager passed on the command line) and then spawns a manager thread. From here on, the application is under the direct control of the thread which provides the necessary virtualisation functionality. The thread intercepts system calls, traps or generates signals and optionally manipulates data being passed to the kernel. When a new application is forked, a new controller thread is spawned. When the application terminates, so does the controller thread. In other words, no program launched from the window manager gets a free rein; it is at the command of its controller thread. All the while, the *Virapix* main process waits for the completion of the first thread and then arranges for the graceful termination of any remaining child processes.

Figure (1)



An Object Oriented View

Virapix takes an object oriented design approach: the core functionality and behaviour is represented by a set of well defined classes. Keeping in mind the low level nature of the software, needless accessor functions are avoided. Likewise functionality not deemed suitable in an object is implemented through global routines. In this subsection, we'll take a comprehensive look at the class organisation and understand what each class does. See Virapix class diagram for a pictorial representation.

class CApp

This is the application class which serves as the entry and exit point for *Virapix* proper. It gathers the necessary information from the OS, sets up the execution environment, launches the window manager and the controller thread and sets the stage for the state machine to take over. This is a singleton class.

What It Does

- Installs handler for the real-time signal *SIGREGISTER* used by state machines to register with the *CApp* object.
- Determines the new root directory and changes to that directory. All future file references in the virtual environment would be relative to this directory which is set to *<USB mount point>/rt*.
- Reads user configuration data stored in *.virapixrc*
- Initialises the logger.
- Records the offset of entry points *mmap()* (in the dynamic linker) and *setpgid()* (in libc) for both 32 bit and 64 bit modes. More on this here and here.
- Masks real-time signals *SIGWAKEPARENT0* and *SIGWAKEPARENT1* used for inter-thread communication.
- Forks a process and puts it in a new group.
- Hooks the new process to *ptrace and then* starts the window manager (passed on the command line)
- Constructs a *CProcCtrl* state machine that would later spawn the controller thread.
- Waits for the completion of this controller thread and then issues *SIGSHUTDOWN* to the remaining applications.
- Shuts down *Virapix*.

class CProcCtrl

This class implements state machine which runs in a thread. There as many *CProcCtrl* objects as there are applications. It is thus the progenitor of the controller thread mentioned earlier. *CProcCtrl* processes events that it gets from the controlled application while in a well defined state.

What It Does

- Initialises state objects; one for each of the six states.
- Registers with *CApp* asynchronously through SIGREGISTER signal.
- Detaches (i.e. disassociates *ptrace*) from the newly forked process. Remember that at this juncture, we are still in the parent thread which had already hooked the process to *ptrace*. See here and also here.
- Spawns a new thread and attaches it (i.e. associates *ptrace*) to the new process. Henceforth, the process will be under the control of this thread.
- Starts the event loop which receives signals from *ptrace* and then transitions to an appropriate state.

CForkState

It represents the state at the time when a new *child* process (i.e. one launched *after* window manager) is forked.

What It Does

- Creates and cranks a *CProcCtrl* state machine for the new process, similar to what *CApp* does for the window manager.
- Blocks till the new process leads a new group.

CStopState

Implements the state corresponding to a SIGSTOP signal. This signal comes from *ptrace* when a new process is attached and also from within *Virapix* as part of inter-thread communication.

What It Does

- If a new process is being attached, *CStopState* configures *ptrace* to enable events from the process and also to turn on auto-attach for all future child processes. It also sets up *setpgid()* parameters and arranges for it to be executed in the controlled process' context.
- If SIGSTOP has originated from *Virapix*, *CStopState* extracts the data accompanying the signal and takes the appropriate action. At present the only thing it does is set up *mmap()* parameters and arrange for the process to map 4KB memory in its address space.

CSyscallState

This class is used whenever the controlled application enters or exits a system call. Obviously, this is the most frequently entered state and consequently the most action packed though much of the work is delegated to subordinate classes.

What It Does

- Keeps track of entry and exit of system calls.

- If the system call is part of the natural execution sequence of the application, *CSyscallState* simply delegates the task to the contained *CIntercept* object.
- If the system call is the one that was inserted earlier and has returned then:
    - for *mmap()*, the start address of the memory map in the process address space is retrieved and stored in the contained *CIntercept* object
    - for *setpgid()*, the parent thread waiting in *CForkState* is woken up and then *mmap()* is inserted in the application address space

CExecState

This state is entered whenever a process launches a new program.

What It Does

- All it does is send *SIGSTOP* to self (i.e. *Virapix*) with information containing the *mmap()* system call number. This signal would be later picked up by *CStopState* as discussed earlier.

CCloneState

CProcCtrl enters this state whenever a process spawns a new thread.

What It Does

Nothing as of now.

CDefState

This class handles events other than the five explained above.

What It Does

- Ignores *SIGTRAP* because it's almost always spurious: *CStopState* configures *ptrace* so as not to deliver raw *SIGTRAP*.
- Terminates the controlled process if the signal received is *SIGSHUTDOWN* (sent by *CApp* as mentioned earlier).
- Delivers all other signals to the process.

CIntercept

An instance of this class lives in *CSyscallState*. Its responsibilities include invoking the appropriate handler function and controlling the usage of the memory map whose allocation we had triggered.

What It Does

- Logically divides the application memory map into 16 slots each of size 256 bytes and doles out the slots to the interceptor whenever requested.
- Invokes the *CSyscallHandler* (described next) interceptor method appropriate to the system call currently being executed.

CSyscallHandler

This is a singleton class that encapsulates the logic of system call interception. The class is huge: it defines close to 90 functions most of them private. Little wonder much of the activity in *Virapix* happens here. It defines interceptor methods for those system calls that in some way modify the host system. This includes open-for-write operations, file creation, attribute or time-stamp modification etc. There are currently 58 interceptor methods.

What It Does

- Examines system call arguments and changes them if an attempt is being made to modify some file on the host system. The new value contains an equivalent path on the USB key. In other words, the host is shielded from user actions: only the USB key is affected. *This is the basis of application virtualisation.* See here for more information.
- Optionally mirrors directory hierarchies of the host system on the USB drive.
- Optionally replaces system calls that take file descriptor (e.g. fchdir()) with those that take file path instead (e.g. chdir()). This is necessary if the descriptor refers to a file on the host computer opened for modification or whose equivalent exists on the USB drive.
- Keeps track of the current working directory and normalises file paths if required.

CSysCompat

This class facilitates virtualisation of 32 bit applications on x86_64 hardware. The system calls are numbered differently on the two platforms and a mapping is required to refer to the correct function.

What It Does

- Maps 32 bit function numbers to their 64 bit equivalents, in short the very system calls that *CSyscallHandler* manipulates. *Virapix* uses this mapping to translate 32 bit numbers obtained "from" *ptrace* to their native 64 bit counterparts. That's how *CIntercept* is able to find the appropriate handler.
- Reverse maps 64 bit function numbers to their 32 bit equivalents. This is a small set which includes only those system calls that *CSyscallHandler* replaces. In short, this mapping is used by *Virapix* to send system call numbers "to" *ptrace*.

CElfReader

This is a utility class template that reads headers in elf binaries (shared object and executables only) and returns the requested information. The template is specialised on the bitness (32 and 64) of the binary.

What It Does

- Temporarily memory maps the elf file and collects header and symbol related information.
- Returns the offset of the requested symbol.

The Activity View

Refer to Virapix activity diagram for a detailed depiction of what we have seen so far.

Behind The Scenes

This section explains some of the more arcane aspects of the design which lie at the very core of *Virapix*.

Ptrace

ptrace forms the backbone of the virtual machine; a system call which provides much of the functionality so crucial for virtualisation. When an application attaches to *ptrace*, the *Virapix* thread that invoked the system call assumes full control of the application which *ptrace* stops with a “decorated” *SIGTRAP* signal for every event including system call entry/exit, forking of a new process etc. This event is encoded in *SIGTRAP*. *Virapix* is then notified about this signal which it decodes, enters one of the six states, performs the necessary tasks and then issues a *ptrace* command to resume the application. The function that delivers the notification is the venerable *waitpid()*. This forms the basis of the event loop.

ptrace celebrates the *attach* event by sending *SIGSTOP* to the process. It is when *CStopState* takes over and configures *ptrace* as discussed earlier. *Virapix* also uses *ptrace* for following tasks:

- to read information about signals (besides *SIGTRAP*) on their way to the process
- to send signals to the process
- to read process memory
- to write to process memory
- to retrieve register values
- to modify register values

System Call Insertion

This isn’t as eerie as it sounds. What it basically means is invoking a system call in another address space, in our case that of the controlled process. In techno speak this translates to temporarily moving the instruction pointer (IP) to the address of our choice. Since *Virapix* by itself can’t play with somebody else’s memory, it enlists the help of *ptrace* to manipulate the registers of the process and force it to execute the desired code. Again, this isn’t any arbitrary code, but the good old *mmap()* and *setpgid()*. First *CElfReader* gives us the offset of these symbols (in *ld.so* and *libc* respectively). Then we use */proc* facility to get the address of the executable segment of these two libraries in the memory map of the process. We add the offset to this address to get the absolute address of the two functions above. This is how we “insert” the system call. *Virapix* uses *ld.so* (instead of the more apt *libc.so*) for *mmap()* because it’s the only library available during the initial stages of program loading. And this is precisely when we need the memory map in the controlled process.

Separate Process Group

When *waitpid()* is passed a negative *pid* (process id), it delivers events from *all* the processes belonging to the group *pid*. Since Linux treats threads very much like processes, we exploit this property to receive events from a multi-threaded process. Unfortunately this also means that *Virapix* will get events from a child process as well! After all, when a process is forked, it belongs to its parent’s group by default. Since we want to track each process in a separate state machine, we should filter out any events coming from other processes. And this is possible only if a process leads its own group. By inserting *setpgid()* as explained above, we meet our objective.

Scratch-pad Memory

Virtualisation is achieved by manipulating system call arguments so that these functions are directed to files on the USB key instead of the files on the host system. Very neat, but how is this achieved? It’s tempting to manipulate arguments in-place, but this would be catastrophic with strings if the new file path happens to be longer than the old. Buffer overruns and what not! The solution is to reserve a dedicated “scratch-pad” memory in the process address space where we can put the new value and pass this address to the system call instead. *Virapix* inserts *mmap()* in the target context which then sets aside the memory for us to use. To keep things simple, light and efficient, *Virapix* requests a page (4KB) large enough to store 16 buffers each of 256 bytes. This places an upper limit (255) on the path length and also on the number of simultaneous access requests (16) to the buffers. Experiments show that this ceiling is sufficiently high.

System Call Manipulation

The algorithm to determine whether or not file path arguments to a system call should be modified is relatively involved, but in simple terms the answer is yes if:

- an existing file on the host computer is being opened in write mode
- the system call is attempting to modify attributes of a file on the host
- a file with the same name exists on the pen drive and at the same location
- a file is being created on the host and the path leading to the file is valid
- the file resides in the user home directory
- the path does not exist on the host but it does on the pen drive

The user home directory needs special consideration. Most applications (*Firefox*, *Open Office* to name a few) store user configuration in the home directory. To avoid a clash (and a potential application breakdown) between the configuration stored on the host and that on the pen drive, *Virapix* always points the system call to the home directory on the USB key whether or not the path exists. If it doesn't the application typically creates it.

Packaging

The software is a suite of products that includes rpm packages for *Virapix*, *Fluxbox Window Manager, Mozilla Firefox, Open Office, Gvim* and of course the install scripts. All are clubbed together in an *iso* file system. The scripts perform an exhaustive set of validations before firing off the actual installation process. Most importantly to meet the demands of virtualisation software, the *rpm* program is invoked with a special set of parameters to target the pen drive and not the host system. In addition, all the post and pre install and uninstall scripts of the packages are pulled out and are invoked separately because of the change of the installation root directory.

TABLE OF CONTENTS