19

Virtual File Systems and Tclkits

The Virtual File System, or VFS, abstraction allows applications to view structured data as a hierarchy of directories and files even though it may not stored as such in a real file system. The data can then be operated on with the standard Tcl channel I/O commands. An example would be a VFS for a remote FTP site that would permit the application to open and perform I/O on a remote file in the same manner as a file on the local system.

Tcl's VFS framework also forms the basis of *tclkits*, which is a technology for deploying entire Tcl applications or packages as a single file without needing any installation steps or additional support files. This chapter describes the use of tclkits as well.

19.1. Using VFS

Although VFS itself is implemented within the Tcl core, accessing it at the script level requires the `tclvfs` extension. It permits both implementation of new VFS types in pure Tcl scripts as well as providing a number of VFS implementations for FTP, WebDAV, ZIP archives and others.

We first demonstrate the use of a VFS through the FTP based VFS available as part of the `tclvfs` extension.

```
% package require vfs::ftp
→ 1.0
```

The next step is to *mount* the VFS at a *mount point* which can be any file system path. By convention, `tclvfs` based packages provide a `Mount` command for this purpose.

```
Mount VFSPATH LOCALPATH
```

Here *VFSPATH* identifies the resource of interest within the VFS and *LOCALPATH* identifies a local file system path where the VFS resource will be made accessible. *LOCALPATH* need not exist and if it does, the file or directory corresponding to the path will not be accessible until the VFS is unmounted. The return value from the `Mount` command is a handle that is later required for unmounting the VFS.

In our FTP example, *VFSPATH* will identify a remote FTP directory.

```
% set mount_handle [vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim]
→ 9
```

We can now treat the remote FTP directory like a local directory modulo some caveats that we list later. For example, we can even change the current directory to that location and list files `glob` or use file system inspection commands.

```
% cd /ftp-vim
% glob *
→ ftp mirror pub site vol
% file size pub/documents/published/books/stevens.netprog.errata.gz
→ 5885
```

We can open channels to files in the VFS and perform I/O on them, **even using channel transforms if desired.**

```
% set fd [open pub/documents/published/books/stevens.netprog.errata.gz rb]
→ rc6
% zlib push gunzip $fd
→ rc6
% gets $fd
→ --------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% gets $fd
→ (Last updated March 19, 1995)
% close $fd
```

When a VFS is no longer required it should be unmounted. Again by convention each VFS supplies a command named `Unmount` to do the needful.

```
Unmount MOUNTHANDLE LOCALPATH
```

The *`MOUNTHANDLE`* argument passed to `Unmount` is the return value from the `Mount` command and *`LOCALPATH`* is the local mount point at which VFS resides. We can unmount our remote FTP directory as follows. Before we do that though, we will switch back to our original directory.

```
% cd ..
% vfs::ftp::Unmount $mount_handle /ftp-vim
→ 1
```

We could also have unmounted the VFS by calling the generic `vfs::unmount` command.

```
vfs::unmount /ftp-vim
```

In that case the mount point suffices to unmount and we do not really require the mount handle.

In general, using `cd` to change the current directory to one located in a VFS is not a good idea. It creates a "split" view of the current directory since the operating system itself is unaware of the existence of the VFS. In any case, changing directories at any time, except possibly during application startup, is a bad idea even outside a VFS since it has process-wide effect.

At this point, we really need to step back and give the Tcl I/O system a big round of applause. Consider what we just did. The "application" code treated a **compressed**, **remote FTP** resource as though it were a plain old local uncompressed file. That is pretty, well, cool!

Before moving on, there are some additional finer points to be noted about VFS.

- A VFS is mounted **process-wide**, meaning all Tcl interpreters within the process see the mount.
- All C code that uses the Tcl file system or channel API's will also see VFS mounts. This means that extensions that use these API's (and not the C runtime routines) will also benefit from being able to treat VFS resources as normal files without any additional coding. So for example, Tk could display images from a VFS exactly as it would from a local file.

By the same token, there are also some caveats,

- A VFS is invisible to the operating system, other processes and any code even within the same process that does not use the Tcl file system API's. For example, you cannot pass an executable stored in a VFS to the `exec` command to run it since the operating system is unaware of the VFS. Note however, that Tcl's `load` command is VFS-aware so you can load a shared library from a VFS. Tcl will copy the shared library to a temporary location on the local file system and pass that path to the operating system loader.

- There are unsurmountable differences between different VFS types that you have to be aware of. For example, not all may support links or may differ in terms of their case-sensitivity (consider a remote FTP VFS on Unix accessed from a Windows client).
- VFS implementations may have some performance-related aspects you need to be aware of. For example, reading even a single byte from a FTP based VFS will cause the entire remote file to be retrieved and loaded into memory.

Even with these caveats, for the most part VFS provides a very generalized way of accessing different resources with the standard Tcl file system and channel commands.

19.1.1. URL mounts

An alternative way of mounting a VFS is based on a URL type. Naturally, this is only possible for virtual file systems based on URL's like FTP or HTTP. The `vfs::urltype` package implements this functionality.

```
% package require vfs::urltype
→ 1.0
```

As before, the `Mount` command is used to mount a specific URL type. It takes a single parameter, the URL type.

```
vfs::urltype::Mount URLTYPE
```

So running the following command

```
% vfs::urltype::Mount ftp
→ Mounted at "ftp://"
```

will result in Tcl treating `ftp://` as an additional volume. We can verify this as below.

```
% file volumes
→ ftp:// C:/ D:/ E:/ F:/
% file split ftp://foo/bar
→ ftp:// foo bar
```

Now **any** FTP based URL can be treated as just another file path. So we could have written our previous example using this method as well.

```
→ set fd [open ftp://ftp.vim.org/pub/documents/published/books/stevens.netprog.errata.gz rb]
  rc8
% zlib push gunzip $fd
→ rc8
% gets $fd
→ ------------------------------------------------------
% gets $fd
→ Typos and errors found in "UNIX Network Programming"
% close $fd
```

When unmounting a `urltype` VFS, simply specify the URL type which serves as the mount handle as well.

```
% vfs::urltype::Unmount ftp
% file volumes
→ C:/ D:/ E:/ F:/
```

19.2. Implementing a VFS

We now turn our attention to implementing a new type of VFS. We will create a VFS that acts as an in-memory file system. Since our purpose is to describe the `tclvfs` interfaces, our virtual file system is very simplistic.

As for channel transforms and reflected channels, implementing a VFS involves writing a handler that implements a set of subcommands for the operations defined by the Tcl file system component. All subcommands take the same first three arguments, shown as *`ROOT`*, *`RELPATH`* and *`ORIGPATH`* in the table below. Here *`ROOT`* is VFS mount path. In our example in the previous section, this would be `/ftp-vim`. *`RELPATH`* is the rest of the path (relative to *`ROOT`*) so that *`ROOT/RELPATH`* is the full absolute path. *`ORIGPATH`* is the path as originally specified in the invocation of the Tcl I/O command. This is translated to *`ROOT/RELPATH`* through normalization.

The subcommands that need to be implemented are shown in Table 19.1.

Table 19.1. VFS driver subcommands

Subcommand	Description
`access` *`ROOT RELATIVE ORIGPATH MODE`*	Called to check if the specified access mode is compatible with permissions on the given path.
`createdirectory` *`ROOT RELATIVE ORIGPATH`*	Called to create a directory.
`deletefile` *`ROOT RELATIVE ORIGPATH`*	Called to delete the specified file.
`fileattributes` *`ROOT RELATIVE ORIGPATH ?INDEX? ?VALUE?`*	Called to retrieve file attribute names or set their values.
`matchindirectory` *`ROOT RELATIVE ORIGPATH PATTERN TYPES`*	Called to retrieve files matching the specified pattern and type.
`open` *`ROOT RELATIVE ORIGPATH`*	Called to open a channel to the specified file.
`removedirectory` *`ROOT RELATIVE ORIGPATH RECURSIVE`*	Called to delete the specified directory.
`stat` *`ROOT RELATIVE ORIGPATH`*	Called to retrieve file information.
`utime` *`ROOT RELATIVE ORIGPATH ATIME MTIME`*	Called to set the access and modification time of a file.

We will now provide an example implementation of a virtual file system. We will implement the commands shown in the table above and make use of the `vfs::filesystem mount` and `vfs::filesystem unmount` commands for mounting and unmounting our file system.

Our `memfs` package will implement an in-memory virtual file system. Mounting a `memfs` will create a new VFS instance with no content. Any content written to this file system will be erased when the VFS instance is unmounted. An application may mount multiple `memfs` VFS instances, each independent of the others.

We require the `vfs` package which exposes Tcl's VFS features at the script level. We will also need the `tcl::chan::variable` package which we will use to implement channels targeting our VFS.

```
package require vfs
package require tcl::chan::variable
```

We will choose to implement our VFS using namespaces as we did for our virtual channel example and define an ensemble corresponding to the VFS driver subcommands.

```
namespace eval memfs {
    namespace ensemble create -parameters {fs_id} -subcommands {
        access createdirectory deletefile fileattributes
        matchindirectory open removedirectory stat utime
    }
}
```

A minor point is worth noting about our namespace ensemble definition. Since we support multiple instances of our VFS, we need to pass an instance identifier to these subcommand procedures in addition to the arguments passed by the VFS core. The VFS core takes a single command prefix and appends its own arguments to it. Our instance identifier will be part of the command prefix and hence will appear **before** the subcommand. We define the ensemble accordingly with the `-parameters` option to indicate the subcommand position. See Section 12.6 for details.

19.2.1. Signalling VFS errors

Let us first deal with signalling of errors as the VFS subsystem expects implementations to use a specific call for this purpose as opposed to directly raising exceptions using the standard Tcl `error` or `throw` commands. This is to ensure a consistent set of error messages and error codes irrespective of the underlying file system.

Errors within the VFS driver should be signalled by calling the `vfs::filesystem posixerror` command and passing it a numeric code representing a POSIX error. This command will then raise a standard Tcl exception with an appropriately formatted error code. Since numeric codes are hard to remember, we will define a wrapper, `posix_error`, that will also accept the mnemonic equivalent of numeric error codes.

```
proc memfs::posix_error {err} {
    if {![string is integer -strict $err]} {
        set err [::vfs::posixError $err]
    }
    vfs::filesystem posixerror $err
}
```

So for example, we can call

```
posix_error ENOENT
```

to report an error that a file or directory does not exist.

19.2.2. Mounting and unmounting

Following `vfs` package conventions, we will name our command for mounting a `memfs` VFS as `Mount`.

```
proc memfs::Mount {mount_path} {
    set id [init_fs]
    vfs::filesystem mount $mount_path [list [namespace current] $id]
    vfs::RegisterMount $mount_path [list [namespace current]::Unmount $id]
    return $id
}
```

It takes a single argument which is the mount point where our file system will be located. It calls an internal command `init_fs` to create a new file system. The `vfs::filesystem mount` command then mounts the file system at the specified location. It has the general syntax

```
vfs::filesystem mount ?-volume? PATH CMDPREFIX
```

Here *PATH* is the mount point and *CMDPREFIX* is the command prefix that implements the VFS driver. In our example, this is the ensemble command we defined earlier which has the same name as our implementation namespace. Notice we also pass in the `memfs` instance identifier returned by `init_fs` command. This allows our implementation to distinguish between multiple `memfs` instances.

The `-volume` prefix specifies that the file system is also a new volume as seen by the `file volumes` command. We saw an example of such a VFS with the `vfs::urltype::ftp` example earlier. This option must not be specified for file systems, like `memfs` that will mount within an existing native file system path.

The call to `vfs::RegisterMount` is strictly not necessary. However, it allows the application to unmount out VFS by passing the mount point to the generic `vfs::unmount` command instead of having to call our VFS-specific `Unmount` command.

Next we implement the corresponding command for unmounting a `memfs` VFS instance.

```
proc memfs::Unmount {fs_id mount_path} {
    variable file_systems
    if {![info exists file_systems($fs_id)]} {
        return
    }
    vfs::filesystem unmount $mount_path
    unset file_systems($fs_id)
    namespace delete $fs_id
    return
}
```

The main point to note here is the call to `vfs::filesystem unmount`. The rest of the code is specific to our implementation and essentially deletes all data associated with that VFS instance. Instead of directly calling our `Unmount` command, it is recommended applications should call `vfs::unmount` so that it can update its database of mounted file systems.

In any case, applications should not call the `vfs::filesystem unmount` command themselves. That command will remove the file system from Tcl's view but will **not** notify the VFS driver that the file system has been unmounted.

19.2.3. VFS operations

We now move on to implementation of the driver commands called by the Tcl VFS subsystem for operating on a file within the VFS. The first four arguments to all these commands are identical. The first is the `memfs` VFS instance id (which we passed as part of the command prefix), the mount point path, the relative path within the VFS, and the original path as specified in the command that invoked the file operation. For example, if our VFS was mounted at `/tmp/mem` and the current working directory was `/tmp`, then a command like

```
file exists mem/foo.txt
```

would result in the arguments after the `memfs` instance id being `/tmp/mem`, `foo.txt` and `mem/foo.txt` respectively.

19.2.3.1. Checking access: `access`

We start off with implementation of the `access` command. This is used by the VFS subsystem to check if a specific file or directory can be accessed with the specified mode. The mode is passed as an additional argument in the form of a bitmask that indicates the type of desired access. If 0, only existence of the file is to be checked. The low three bits signify execute (least significan bit), write and read access respectively. Our file system does not implement access permissions and so for directories we will allow all access types. For files however, we will disallow execute access since the operating system has no knowledge of our file system and cannot execute files residing in it. (At the Tcl level, `exec` will not work.)

```
proc memfs::access {fs_id root relpath origpath mode} {
    switch -exact -- [node_type $fs_id $relpath] {
        "" { posix_error ENOENT }
        file { if {$mode & 1} { posix_error EACCES } }
        dir { }
    }
    return
}
```

The implementation uses our internal `node_type` command to check for whether the path is a regular file or directory. If the specified access is allowed the command returns normally with the return value being immaterial. If access is **not** allowed, the command must raise a POSIX error as discussed previously. In our case, we return `ENOENT` when the path does not exist and `EACCES` when it does not have the requested execute access permission.

19.2.3.2. Creating directories: `createdirectory`

Next we implement the `createdirectory` call which is straightforward. The command is expected to create a new directory at the specified path within our file system if it does already exist. If the path corresponds to an existing regular file, it should raise a POSIX error.

```
proc memfs::createdirectory {fs_id root relpath origpath} {
    node_add_dir $fs_id [node_find $fs_id $relpath dir]
}
```

The implementation uses two internal commands. The first of these, `node_find`, maps a file path to the location key for the corresponding node in our internal file system structures. The node itself need not exist but if it does, the optional third argument mandates that it must be of the specified type. Thus in the above call, `node_find` would raise a POSIX error if the node existed and was not a directory. We will see `node_find` used throughout our implementation.

The other internal command, `node_add_dir` simply creates an empty directory and associated structures at a specified node location.

19.2.3.3. Deleting directories: `removedirectory`

The `removedirectory` command deletes a directory. It takes an additional argument which must be a boolean value. If `true` then the directory and its contents should be recursively deleted; otherwise the command should raise a POSIX error if the directory is not empty. It is not an error if the directory does not exist.

```
proc memfs::removedirectory {fs_id root relpath origpath recursive} {
    node_del_dir $fs_id [node_find $fs_id $relpath dir] $recursive
}
```

The implementation is pretty much identical to that of `createdirectory` above except we call `node_del_dir` which does the hard work of deleting the directory structure and its contents.

19.2.3.4. Creating and opening files: `open`

The `open` VFS driver subcommand essentially has to implement the file system level operations of the Tcl `open` and `chan open` commands which applications use to create files as well as open them for I/O. The command takes two additional arguments that specify the access mode and permissions in the case of creating a new file. These are equivalent to the ones passed to the Tcl `open` command.

```
proc memfs::open {fs_id root relpath origpath mode perms} {
    variable file_systems

    set node_key [node_find $fs_id $relpath file]
    set exists [expr {[node_type $fs_id $relpath] ne ""}]
    set truncate 0
    switch -glob -- $mode {
        "" -
        r* {
            if {! $exists} {
                posix_error ENOENT ❶
            }
        }
        a* -
        w* {
```

```
            if {$exists} {
                if {[string index $mode 0] eq "w"} {
                    set truncate 1
                }
            } else {
                node_add_file $fs_id $node_key
            }
        }
        default {
            error "Unsupported mode \"$mode\""
        }
    }
    set chan [node_add_channel $fs_id $node_key $truncate]
    set close_callback [list [namespace current]::node_close_handler  \
                            $fs_id $node_key $chan]
    return [list $chan $close_callback]
}
```

❶ File must exist

The implementation of open is a little longer than others but should be straightforward to follow at this stage. We have already described the `node_find` and `node_type` internal commands. The `mode` parameter, as for the Tcl open, specifies the open mode in the form of `r`, `r+`, `w` etc. and the `switch` statement takes appropriate action depending on the mode. The `node_add_file` command creates a new node of type file in our file system structure.

The return value of the command should be a list of one or two elements. The first element must be the handle to an open channel to use for performing I/O on the file. The second element is optional. If present it should be command prefix to be invoked when the channel is closed. Our internal `node_add_channel` command creates a new channel to a specified file (node). It also adds the channel to the list of channels internally associated with the file since we want to prevent files from being deleted while they have channels open to them. For the same reason, we also need to know when a channel is closed so that we can remove it from this list. Thus we make use of the second optional element in the return value to declare a callback to be invoked on channel close. This callback `node_close_handler` will remove the channel from the channel list and also update the access and modification time stored for the node (file).

19.2.3.5. Deleting files: `deletefile`

The `deletefile` command is almost identical to the `removedirectory` command we implemented above except it deals with removal of files, not directories, and hence has no need for recursion control.

```
proc memfs::deletefile {fs_id root relpath origpath} {
    node_del_file $fs_id [node_find $fs_id $relpath file]
}
```

A design decision we have made is that files with open channels to them cannot be deleted. This follows the Windows model (as opposed to Unix). Our `node_del_file` implementation will report an error via `posix_error` on an attempt to delete a file which is open. Naturally, this decision affects `removedirectory` as well.

19.2.3.6. Setting file timestamps: `utime`

The `utime` command is called to set the last access and modification timestamps for a file or directory. It takes two additional arguments corresponding to the access and modification time respectively. These are specified in terms of the number of seconds since the epoch, January 1, 1970.

```
proc memfs::utime {fs_id root relpath origpath atime mtime} {
    node_set_times $fs_id [node_find $fs_id $relpath] $atime $mtime
}
```

The internal commands in our implementation set these timestamps for various operations. For example, creation of a file will also update the modification time for its parent directory. The `utime` command is specifically called by Tcl in response to the `file atime` and `file mtime` commands.

19.2.3.7. File statistics: `stat`

The `stat` command is returns information about a file or directory. The return value from the command should be a dictionary with the following keys: `dev`, `ino`, `mode`, `nlink`, `uid`, `gid`, `size`, `atime`, `mtime`, `ctime` and `type`. These keys have exactly the same semantics we described for the `file stat` command.

```
proc memfs::stat {fs_id root relpath origpath} {
    return [node_stat $fs_id [node_find $fs_id $relpath]]
}
```

Our node implementation maintains the relevant information internally and our driver API can again just delegate to it.

19.2.3.8. File attributes: `fileattributes`

As discussed in the Files and Basic I/O chapter, files can be associated with certain attributes that are specific to the file system. These attributes are managed with the `file attributes` commmand. VFS drivers need to implement a corresponding `fileattributes` command to allow Tcl to access attributes for files within the VFS. The command is invoked in three forms:

- If no additional arguments (other than the standard ones) are specified, the command should return a list of attribute names supported by the file system. **Every such call must return the same names in the same order.**
- If a single additional argument is specified, it will be an integer index into the list of attribute names. The command should return the value of this attribute.
- If two additional arguments are specified, the first is the integer index into the attribute list as above. The second is the value to assign to the attribute.

Our implementation is shown below.

```
proc memfs::fileattributes {fs_id root relpath origpath args} {
    set attr_names [lsort [node_attr_names]]
    if {[llength $args] == 0} {
        return $attr_names
    }
    set node_key [node_find $fs_id $relpath]
    set attr_name [lindex $attr_names [lindex $args 0]]
    if {[llength $args] == 1} {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name]
    } else {
        return [node_attr $fs_id [node_find $fs_id $relpath] $attr_name [lindex $args 1]]
    }
}
```

As always, it relies on the underlying internal functions to to the actual work. The `node_attr_names` command returns a list of attribute names supported by our VFS. We ensure we always pass them back in the same order by sorting them before returning. We then use the `node_attr` command to get or set the attribute on the node corresponding to the specified path.

Although not shown above, our VFS supports two attributes: `-contenttype` and `-encoding`. Applications can assign any values they want to these attributes as VFS has no interest or control over their semantics. The **intended** use is for applications to store the encoding used for the file content in the `-encoding` attribute and the content type (similar to the `Content-Type` HTTP header) which specifies the format of the content like `text/html`

in the `-contenttype` attribute. Note these are advisory attributes and have to be explicitly set by the application There is no way for the VFS to detect the encoding in use or the type of content stored in the file.

19.2.3.9. Matching files: `matchindirectory`

One final VFS driver command remains to be implemented. The `matchindirectory` command is what Tcl's file system calls to retrieve directory contents. The return value of the command is expected to be a (possibly empty) list containing file names. Commands like `glob` also support matching based on patterns and file types. Correspondingly, the `matchindirectory` command takes two additional arguments that specify a glob pattern and a file type specifier.

If the glob pattern is the empty string, the `relpath` argument is the relative path of a file or directory. The command should then only check if the path exists **and** is of the specified type. If so, it should return a list containing the corresponding **original path** that was specified on the command line, not the relative path. If the path does not exist or is not of the specified type, an empty list is returned.

If the glob pattern is not the empty string, the `relpath` is always a path to an existing directory whose contents are to be matched against the pattern. The command should then return the names that match the pattern and the specified type.

```
proc memfs::matchindirectory {fs_id root relpath origpath pat type} {
    variable file_systems
    if {[string length $pat] == 0} {
        set file_type [node_type $fs_id $relpath]
        if {($file_type eq "dir" && [::vfs::matchDirectories $type]) ||
            ($file_type eq "file" && [::vfs::matchFiles $type])} {
            return [list $origpath]
        } else {
            return {}
        }
    }

    if {[node_type $fs_id $relpath] ne "dir"} {
        return {}
    }

    set node_key [node_find $fs_id $relpath]
    set matches {}
    if {[::vfs::matchDirectories $type]} {
        foreach name [node_subdirs $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    if {[::vfs::matchFiles $type]} {
        foreach name [node_files $fs_id $node_key] {
            if {[string match $pat $name]} {
                lappend matches [file join $origpath $name]
            }
        }
    }
    return $matches
}
```

Our implementations makes use of two internal commands `node_files` and `node_subdirs` that return the names of files and subdirectories under the specified node. These are then matched against the pattern to construct the returned list.

The `type` argument specifies whether the returned list should include files, directories or both. We treat it as opaque and use the utility commands `vfs::matchDirectories` and `vfs::matchFiles` to ascertain whether

entries of a specific type are to be included or not. Note that the type argument can indicate that **both** files and directories are to be included.

The Tcl `glob` command allows the type argument to limit matching files based on other criteria as well such as permissions. These criteria are not exposed for VFS systems.

The `vfs` package provides another utility command `vfs::matchCorrectTypes` that serves as an alternative to `matchDirectories` and `matchFiles`.

```
vfs::matchCorrectTypes TYPES FILELIST ?DIR?
```

The command returns a list of names from *`FILELIST`* that fulfil the type requirements specified by *`TYPES`*. If *`DIR`* is not specified, *`FILELIST`* must contain absolute paths. If *`DIR`* is specified, it must be a directory and *`FILELIST`* should be the list of file and directory names within that directory. Depending on your internal file system implementation, you may find `matchCorrectTypes` more convenient to use.

19.2.3.10. `memfs` internals

Due to space limitations, we will not go into detail regarding our `node_*` commands that implement our VFS internals. You can download the `memfs.tcl` file from the book's web site to see the implementation.

Time to see if our VFS actually works.

```
% package require memfs
→ 1.0
% memfs::Mount /mem
→ 1
% file mkdir /mem/dir
% set fd [open /mem/dir/foo.txt w]
→ rc24
% puts -nonewline $fd "It lives!"
% set enc [fconfigure $fd -encoding]
→ cp1252
% close $fd
% glob /mem/dir/*
→ C:/mem/dir/foo.txt
% file attribute /mem/dir/foo.txt -contenttype text -encoding $enc
```

It appears the file was created. Let us read it back. We stored the encoding used to write it as a file attribute. So we will configure the channel accordingly while reading.

```
% set fd [open /mem/dir/foo.txt]
→ rc25
% fconfigure $fd -encoding [file attribute /mem/dir/foo.txt -encoding]
% read $fd
→ It lives!
```

An attempt to delete the file should fail as we have the file open. Retrying after closing the file should allow the delete operation to succeed.

```
file delete /mem/dir/foo.txt Ø error deleting "/mem/dir/foo.txt": permission denied
file exists /mem/dir/foo.txt → 1
close $fd                    → (empty)
file delete /mem/dir/foo.txt → (empty)
file exists /mem/dir/foo.txt → 0
```

Finally we verify we can unmount our file system.

```
% vfs::unmount /mem
% file exists /mem
→ 0
```

It's all good! Ship it!

19.3. VFS introspection

The set of current VFS mounts can be retrieved with the `vfs::filesystem info` command. With no arguments, the command returns the list of mount points.

```
% memfs::Mount /mem
→ 1
% vfs::ftp::Mount ftp://ftp.vim.org /ftp-vim
→ 0
% vfs::filesystem info
→ C:/ftp-vim C:/mem
```

If the optional argument is specified, it must be a mount point path. In this case the command returns the VFS driver command prefix that will be invoked to handle requests to that file system.

```
% vfs::filesystem info /mem
→ ::memfs 1
% vfs::filesystem info /ftp-vim
→ vfs::ftp::handler 0 {}
```

19.4. Single file deployment: Tclkit, Starkit, Starpack

Except in the simplest cases, deploying an application or even a large library package involves distribution of

- The Tcl interpreter itself, e.g. `tclsh` or a custom built executable
- The Tcl script files comprising the application
- Any packages, modules and binary extensions required
- Support files such as icons and other resources

The presence of multiple files in a directory hierarchy means deployment cannot be a simple copy and run operation. The files have to combined into some archive or installation format for distribution and then unpacked or installed on the target system. Although this can be an inconvenience, there is another more irksome issue for the user. If multiple unrelated Tcl applications are installed, there is potential for interference between the applications. Each may require different versions of Tcl and libraries, expect different settings in environment variables like `TCLLIBPATH` and so on.

The Tclkit technology is a solution that makes deployment as simple as copying a single file and running it with no additional steps required. Moreover, the application is completely self-contained and will have no interference with other Tcl installations including other Tclkit based applications.

> **Single file deployment alternatives**
>
> There are several alternative solutions for single file deployment similar to that of Tclkit. Here we describe Tclkit because it is probably the most widely used and also the one with which the author is most familiar. However, other solutions may have features not present in Tclkit based applications. For example, Freewrap can optionally encrypt the contents of the deployed file.

19.4.1. Tclkits, starkits, starpacks

A *starkit* (**St**and**a**lone **R**untime) is a way to package an entire directory hierarchy and its contents into a single file. Conceptually starkits are similar to file archival formats such as `tar` or `zip` and in fact there are starkit variations based on those archive formats as well. What sets the starkit apart is some additional internal structure that allows a starkit to be mounted as a VFS.

The internal format of starkit archives was originally based on a database called Metakit. There are now also variations that use other formats like the above-mentioned zip format. For our purposes, the internal formats are immaterial for the most part and we will refer to them collectively as starkits.

The `tclkitsh` and `tclkit` applications are specially enhanced versions of `tclsh` and `wish` respectively that understand the format of Metakit-based starkits. The other formats of starkits have their own corresponding applications. For example, the applications that understand the Vlerq format are `tclkit-cli` and `tclkit-gui`. All these applications, which we will collectively refer to as *tclkits*, mount the starkit file as a VFS.

The tclkit programs can also be used in place of the `tclsh` and `wish` shells. See Section 19.4.3.

Starkits and tclkits together comprise a two-file solution for distributing Tcl applications. So, for example, an application packaged as a starkit `myapp.kit` can be run as

```
tclkitsh myapp.kit ARG …
```

However, we can go a step further. The tclkit and starkit can be further combined into a **single** executable file, termed a *starpack*. This is a self contained executable holding the Tcl interpreter and libraries as well as as the application code with its supporting files. A starpack `myapp` (`myapp.exe` on Windows) constructed from the `tclkitsh` and `myapp.kit` above can be executed simply as

```
myapp ARG …
```

19.4.2. Obtaining a tclkit

Tclkits are available from multiple sources on the Internet. We list only some of the more popular ones below. Although originally based the same technology, they have some differences in their internal structure and build systems.

19.4.2.1. Downloading prebuilt tclkits

The easiest way to obtain a prebuilt tclkit is to download the appropriate version for your operating system platform from one of the locations below.

- The KitCreator[1] project
- The Kitgen Build System[2] (KBS) project
- The ActiveState[3] distribution includes tclkit binaries. They are termed *basekits* and available in the `bin` directory of an ActiveTcl installation.

19.4.2.2. Building tclkits

Both KitCreator and KBS also provide build scripts that allow you build your own tclkit executables in case the prebuilt binaries do not support your platform or you want a custom version with a different internal format or

[1] http://tclkits.rkeene.org
[2] https://sourceforge.net/projects/kbskit/files/kbs
[3] http://tcl.activestate.com

additional packages. Both download the required source files from their repositories. Building a tclkit then simply involves invoking the appropriate script, `kitcreator` in the case of KitCreator and `kbs.tcl` in the case of KBS.

KitCreator has two additional features:

- It supports cross-compiling a tclkit for a different target platform than the one on which the build system is running.
- It has an online build system[4] where you can specify through a Web interface one of almost two dozen target platforms and select any additional extensions desired. It will then build a customized tclkit for download.

Both KitCreator and KBS are based on the GNU toolchain. For building on Windows you will need to install MinGW[5] or MinGW-w64[6]. Alternatively, for Microsoft Visual C based builds, you may prefer to instead use the original kitgen[7] system. In addition to the GNU toolchain, this also provides `nmake` based makefiles suitable for Visual C builds. See the `README` file at the toplevel for instructions.

Our examples below assume we are running a downloaded tclkit that we have renamed as `tclkit-cli.exe`.

19.4.3. Using tclkits as Tcl shells

Tclkit binaries can be for the most part used in place of the standard Tcl shells `tclsh` and `wish`. In particular,

- when run without arguments, they will display an interactive prompt and execute any commands entered
- you can pass a Tcl script file to run and additional arguments on the command line just as for the standard shells

These characteristics make it very convenient to use tclkit interpreters on systems where Tcl is not installed. You can copy a single executable and gain full use of a Tcl command shell.

There are however some differences.

- In addition to Tcl script files, tclkit programs will also execute starkits as we explore in a bit. As of Tcl 8.6, the standard shells do not have the requisite VFS drivers built-in and will not recognize the starkit formats.
- Because tclkits are self contained, they do not examine environment variables like the `TCLLIBPATH` environment variable when setting up the `auto_path` variable for locating packages.
- Some tclkit variations do not include a full set of time zone and character encoding data. If this matters to your application, it is simply a matter of downloading or building one that does include this data.

19.4.4. The `sdx` tool

As we described previously, a starkit is a directory tree and its content packaged as a single file. Although a starkit can be constructed using Tcl's base VFS facilities, most commonly the `sdx` tool is used for this purpose. This wraps all the initialization and low level operations required to build a starkit into a set of high level commands callable from a command line. You can download it from several Tcl sites, for example https://chiselapp.com/user/aspect/repository/sdx/index.

Because `sdx` is itself packaged as a starkit, you can only run it using using a tclkit, not with `tclsh`.

The `sdx` tool comes with a built in help system. The general syntax for running the tool is

```
tclkit-cli sdx.kit ?SDXCOMMAND? ?COMMANDARGS?
```

If no arguments are provided, it will print a summary of available commands.

[4] http://kitcreator.rkeene.org/kitcreator
[5] http://www.mingw.org
[6] https://mingw-w64.org
[7] https://github.com/patthoyts/kitgen

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit
→ Specify one of the following commands:
  addtoc eval fetch ftpd httpd httpdist ls lsk md5sum mkinfo mkpack mkshow mksplit mkzipkit
   ↳ qwrap ratarx rexecd starsync sync tgz2kit treetime unwrap update version wrap
  For more information, type:  sdx.kit help ?command?
```

The help command offers more detailed information for each command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit help qwrap
→
   Quick-wrap the specified source file into a starkit

           Usage: qwrap file ?name? ?options?

           -runtime   file   Take starkit runtime prefix from file

           Generates a temporary .vfs structure and calls wrap to create
           a starkit for you.  The resulting starkit is placed into file.kit
           (or name.kit if name is specified). If the -runtime option is
           specified a starpack will be created using the specified runtime
           file instead of a starkit.

           Note that file may be a local file, or URL (http or ftp).
```

You will notice the tool comes with wide-ranging functionality, even including a basic FTP and Web server. Our discussion will be limited to the functionality related to the topic at hand — working with starkits and starpacks.

19.4.5. Building a single script starkit: sdx qwrap

Let us start with the simplest possible example — building a starkit from a single script. We will create a starkit containing the ubiquitous Hello World! program. We first create the file containing our script.

```
% write_file hello.tcl {puts "Hello World!"}
```

We now convert this to a starkit using the sdx qwrap command. At the Windows command prompt,

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl
→ 5 updates applied
```

We see that a starkit hello.kit has been created.

```
c:\temp\tclkit-demo> dir /b *.kit
→ hello.kit
  sdx.kit
```

We can run the created starkit with the tclkit-cli application.

```
c:\temp\tclkit-demo> tclkit-cli hello.kit
→ Hello World!
```

Now, this obviously not very different from running

```
c:\temp\tclkit-demo> tclsh hello.tcl
→ Hello World!
```

But hold on before you thumb your nose at this. We will see next how this can then be used to build a fully self-contained executable.

We will also see later that the starkit is not constrained to contain a single file. An entire directory structure with multiple packages, extensions, and auxiliary files comprising complete application can be included within the starkit.

19.4.6. Building a single script executable: sdx qwrap -runtime

Distributing a starkit implies also distributing a tclkit executable or having the end user obtain it from somewhere. We can do better by combining the tclkit executable and the starkit into a single executable file — a *starpack*.

First, we need to make a copy of our tclkit executable. We will go into the reasons for this when we discuss the structure of tclkits.

```
c:\temp\tclkit-demo> copy tclkit-cli.exe tclkit-cli-runtime
→         1 file(s) copied.
```

We then run the `sdx qwrap` program again but this time with the `-runtime` option.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit qwrap hello.tcl hello.exe -runtime tclkit-cli-runtime
→ 872 updates applied
  4 updates applied
```

And voila, we now have a `hello` program, our very first starpack!

```
c:\temp\tclkit-demo> dir /b hello.*
→ hello
  hello.kit
  hello.tcl
```

Notice a quirk of the `sdx qwrap` command. **Even on Windows the executable file is created without an `exe` extension.** So if you are on that platform, it needs to renamed appropriately.

```
c:\temp\tclkit-demo> rename hello hello.exe
```

We now have a fully functional single file executable that comprises our entire application.

```
c:\temp\tclkit-demo> hello
→ Hello World!
```

The convenience of this cannot be overstated. Deployment consists of copying a file to the target system and installation is a no-op. And as we will see as we proceed through the chapter, these benefits are not limited to toy applications implemented in a single file.

19.4.7. Internal structure of a starkit

The `sdx` utility's `lsk` command allows us to inspect the internal structure of a starkit.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.kit
→
  hello.kit:
                            dir  lib/
          80  2017/07/04 11:45:57  main.tcl

  hello.kit/lib:
```

```
                              dir  app-hello/

  hello.kit/lib/app-hello:
          54  2017/07/04 11:45:57  hello.tcl
          79  2017/07/04 11:45:57  pkgIndex.tcl
```

Moreover, the `lsk` can even list the contents of a starkit embedded in a starpack.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit lsk hello.exe
→
  hello.exe:
        5114  2013/02/16 22:53:30  boot.tcl
          37  2013/02/16 22:53:30  config.tcl
                              dir  lib/
          80  2017/07/04 11:45:57  main.tcl
       57022  2013/02/16 22:53:30  tclkit.ico

  hello.exe/lib:
                              dir  app-hello/
...Additional lines omitted...
```

Notice the contents of the starpack are much larger since it includes not just the `hello` application but also the Tcl runtime. The format of the output should give you a hint that starkits are structured just like file systems and in fact are implemented as a VFS.

We can proceed to examine the internal contents in one of two ways. The first is through the usual VFS and Tcl I/O commands. The second is by extracting the contents of the startkit with the `sdx unwrap` command.

Let us first demonstrate the former, primarily to prove that the starkit is accessible as a VFS. We need to use the `tclkit-cli` application, and not `tclsh` for this as the latter does not by default have the requisite VFS drivers. From within `tclkit-cli` we first load the Metakit VFS driver and then mount the starkit.

```
% package require vfs::mk4
→ 1.10.1
% vfs::mk4::Mount hello.kit /hello
→ mkclvfs1
% glob /hello/*
→ C:/hello/main.tcl C:/hello/lib
```

We see that top level directory of the VFS has two entries, `main.tcl` and `lib`. We can access any file within the VFS with the standard Tcl I/O commands.

The second way to examine the content of a starkit (or starpack) is by extracting its contents with the `sdx unwrap` command.

```
c:\temp\tclkit-demo> tclkit-cli sdx.kit unwrap hello.kit
→ 5 updates applied
```

This will extract the contents of the starkit into a local directory `hello.vfs`.

```
c:\temp\tclkit-demo> dir /s /b hello.vfs
→ C:\temp\tclkit-demo\hello.vfs\lib
  C:\temp\tclkit-demo\hello.vfs\main.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\hello.tcl
  C:\temp\tclkit-demo\hello.vfs\lib\app-hello\pkgIndex.tcl
```

We can take a peek at the content of the `main.tcl` file.

```
c:\temp\tclkit-demo> type hello.vfs\main.tcl
→
    package require starkit
    starkit::startup
    package require app-hello
```

We will leave the details about the commands in `main.tcl` for the next section where we build a more complete example of a starkit. For the moment we will comment on some general points points about the tclkit structure.

The root of the VFS contains a file `main.tcl`. This file will be sourced by the tclkit application when the starkit is loaded. The `main.tcl` file above was automatically created by the `sdx qwrap` command. There is no requirement that this file have exactly the contents shown. It could contain any sequence of commands or even an entire application.

The only mandated requirement for starkits is the existence of this `main.tcl` file at the root of the starkit VFS. The rest of the starkit VFS may be structured in any fashion you choose.

The structure of our `hello` starkit is the boilerplate used by `sdx qwrap` for its automatically generated starkits. It has a `lib` directory beneath which it expects all packages to be placed. The `starkit::startup` command in `main.tcl` will add the directories below this to the `auto_path` variable. In the case of `sdx qwrap`, there is no option for including additional packages so this directory contains only a single subdirectory `app-hello`.

This directory contains our `hello.tcl` script **converted to a package form** and loaded with the `package require` command in `main.tcl`. To convert our script to a package, `sdx qwrap` adds the `pkgIndex.tcl` file

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\pkgIndex.tcl
→
    package ifneeded app-hello 1.0 [list source [file join $dir hello.tcl]]
```

and modifies our script to include `package provide` command.

```
c:\temp\tclkit-demo> type hello.vfs\lib\app-hello\hello.tcl
→ package provide app-hello 1.0

  puts "Hello World!"
```

We reiterate at this point that this structure is completely optional. If we were to manually structure the VFS, something we illustrate in the next section, we could have just put our script file in the VFS root directory and directly sourced it instead of converting it to a package. Or we could have included our script within `main.tcl` itself. The structure used by `sdx qwrap` is reflective of the conventions used when a `starkit` is used to deploy multiple packages and more complex applications.

19.4.8. A more complete starkit example: `sdx wrap`

Having seen the creation of a starkit from a single script, let us now create a more complete example. Our demo starkit will be multifunctional: it will include the `sequences` package from Section 13.3.8 as well as the standard `Hello World!` functionality. We foresee great demand for this combination.

Our demo has certain operational requirements:

- The starkit must be usable as a library where it can be loaded in the main application.
- It should also be usable as a standalone application itself when it is passed as the script argument to a tclkit application. In that case it should run the command specified by the user.
- It should be deployable as a single file executable.
- For ease of development, we should be able to run it in normal fashion even when it is not wrapped into a starkit. That way we can edit the files during development and re-test without having to build a starkit after every change.

We will follow the same basic structure as was created by `sdx qwrap`. At the top level, we have the `demo.vfs` directory which will be the root of our starkit's virtual file system. The directories and files below this are shown below.

```
c:\demo-dir> ls -R demo.vfs
demo.vfs:
hello.tcl  lib  main.tcl

demo.vfs/lib:
app-demo  sequences

demo.vfs/lib/app-demo:
demo.tcl  pkgIndex.tcl

demo.vfs/lib/sequences:
pkgIndex.tcl  seq_arith.tcl  seq_geom.tcl
```

We will start with the mandatory `main.tcl` file in the root of the starkit VFS.

```
# main.tcl
namespace eval demo {}
package require starkit
set demo::run_mode [starkit::startup]
set demo::vfs_root [file dirname [file normalize [info script]]]

package require sequences
source [file join $demo::vfs_root hello.tcl]

puts "run_mode: $demo::run_mode"

switch -exact -- $demo::run_mode {
    sourced { }
    unwrapped -
    starkit -
    starpack {
        package require app-demo
    }
    default {
        error "Unknown run mode $demo::run_mode"
    }
}
```

The scripts creates a namespace `demo` for its own use and then loads the `starkit` package that implements the required VFS drivers built into a tclkit application. The next command is a call to `starkit::startup` which has two effects that are directly relevant to us:

- It initializes the `auto_path` variable used for loading packages to the `lib` directory in the VFS. You are of course free to further modify `auto_path` as appropriate for your application. For example, you might have another directory as a sibling of `lib` that you want to add to the package path.
- It returns a value that indicates how the starkit is being used. We use this to determine whether our starkit should behave as a bundle of packages or a standalone application. The possible return values are shown in Table 19.2.

Specific tclkit variations may return values other than those shown in the table; for example, `service` indicating the starkit is running as a Windows service. See the documentation for your `tclkit` for these additions.

Table 19.2. Starkit start-up modes

Mode	Description
unwrapped	Indicates that the main.tcl is **not** part of a starkit and is being read in as a regular Tcl script. As we will see below, it is convenient during development for the application to be in "unwrapped" form as a set of Tcl scripts instead of a monolithic single-file starkit.
sourced	Indicates that the starkit was loaded with the source command either from the application or another starkit. This would generally indicate that the starkit is not itself the main application.
starkit	The starkit is the main file being run by the tclkit application from the command line and as such should provide the application functionality.
starpack	The starkit is bound to the tclkit executable thereby comprising a single-file application.

We will see all these modes in our example scenarios below.

The script then preloads our starkit functionality. It loads the sequences package which is placed under the lib directory and therefore found via auto_path. We chose not to implement hello as a package so the script simply sources it. Note that instead of preloading these, we could have chosen to omit these lines and thereby leaving it up to the application to explicitly load them.

For the purposes of our demonstration, we then print out the mode that the starkit is running in.

Finally, the main script checks the mode. A value of sourced indicates that the starkit is being loaded as a library. Nothing more needs to be done in this case as we have already loaded the contained functionality. All other values of the mode indicate the the starkit should run as an application. It then loads our application code which is implemented, following starkit conventions, as a package.

Our Hello World! functionality is our simple script from the previous section written as a procedure.

```
# hello.tcl
proc hello {} {
    puts "Hello World!"
}
```

Finally, we come to our application code. It is simple enough that we just present it here without any explanation. Our example usage later will clarify working if needed.

```
# demo.tcl
package provide app-demo 1.0
package require sequences

if {[catch {
    switch -exact -- [lindex $argv 0] {
        hello { hello }
        arith { seq::arith_term {*}[lrange $argv 1 end] }
        geom { seq::geom_term {*}[lrange $argv 1 end] }
        default {
            error "Unknown or missing command: must be hello, arith, geom"
        }
    }
} result]} {
    puts stderr $result
    exit 1
} else {
    if {$result ne ""} {
        puts stdout $result
    }
}
```

We are now ready to actually build a starkit from our demo application. The sdx utility's wrap command will create a starkit from a specified directory tree.

```
tclkit-cli sdx.kit wrap NAME ?options?
```

The sdx wrap command takes several options but we only describe basic usage here. It creates a starkit named *NAME* whose contents are created from a directory of the same name but with a .vfs file extension.

```
c:\demo-dir> tclkit-cli sdx.kit wrap demo -interp tclkit-cli
10 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

The command creates two files, the starkit demo and a Windows batch file demo.bat. The latter is simply a Windows batch file that invokes the tclkit application passing it the starkit that was created.

```
@tclkit-cli demo %1 %2 %3 %4 %5 %6 %7 %8 %9
```

On Windows we can then run the starkit through the demo batch file. On Unix systems, the starkit can be directly run because it begins with the header

```
exec tclkit-cli "$0" ${1+"$@"}
```

causing Unix shells to run the starkit by passing it to tclkit-cli.

The purpose of the -interp option when creating the starkit is to specify which tclkit application should be used to run the starkit when the starkit is directly invoked in the Windows or Unix command shell. We specified it as tclkit-cli as that is the tclkit application we are using. If unspecified, it would default to tclkit.

The starkit can be run using **any** tclkit application by explicitly passing it as the command line argument. The use of the -interp option only applies to the case where the starkit is specified as the program name in the shell command line.

We are now ready to try out our application in various modes. First, we will use it as a package bundle in interactive mode.

```
c:\demo-dir> tclkit-cli
% source demo
run_mode: sourced
% hello
Hello World!
% seq::arith_term 2 3 4
11
```

Notice the run mode is printed as sourced.

Next, we will try running as an unwrapped application.

```
c:\demo-dir> tclkit-cli demo.vfs/main.tcl hello
run_mode: unwrapped
Hello World!

c:\demo-dir> tclkit-cli demo.vfs/main.tcl arith 2 3 4
run_mode: unwrapped
11
```

This allows us to edit and modify the files in the `demo.vfs` tree and test it without having to rebuild the application in test mode.

In production, the starkit will be directly invoked as the application either via passing it to the tclkit application,

```
c:\demo-dir> tclkit-cli demo hello
run_mode: starkit
Hello World!
```

or alternatively via the batch file (on Windows) or directly invoking the starkit (on Unix)

```
c:\demo-dir> demo arith 2 3 4
run_mode: starkit
11
```

The final variation for running our application is as a single-file executable. Like `sdx qwrap`, `sdx wrap` takes a `-runtime` option that specifies a tclkit application to use as the runtime for our starkit. The two are then bound together in a single executable.

We will use a slight variation of `sdx wrap` that uses the `-vfs` option.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime tclkit-cli-runtime
872 updates applied
9 updates applied

c:\demo-dir> ls
demo  demo.bat  demo.vfs  myapp.exe  sdx.kit  tclkit-cli-runtime  tclkit-cli.exe
```

This creates an starpack `myapp.exe`. The `-vfs` option specifies the VFS directory as `demo.vfs` and not `myapp.vfs` which would be the default based on the starpack name.

We now have single file application that can be copied to any Windows system and run without any additional steps.

```
c:\demo-dir> myapp hello
run_mode: starpack
Hello World!

c:\demo-dir> myapp arith 2 3 4
run_mode: starpack
11
```

19.4.9. Considerations for multiplatform starkits

Let us take a moment to talk about issues related to multi-platform support. Starkits which are purely script based and have no binary extensions in their content, can be loaded by any tclkit application on any platform. This is true irrespective of the platform on which the starkit was built. Thus the `demo` starkit we built on our Windows system would work just as well on OS X or Linux.

Starkits that contain binary extensions are also portable across platforms as long as they follow the appropriate structure for packages that we describe in Section 13.7. The `load` command for loading binary extensions will recognize that the extension is in starkit and copy it out to the local file system from where it can be loaded by the OS loader.

Starpacks are platform-specific executables and, by their very nature, are not portable. You need to build a separate starpack for every platform for which you want to deploy a single-file executable.

Even though starpacks are platform-specific, they do not need to be built on their native platform. All you need to do is specify the appropriate tclkit for the target platform as the value of the `-runtime` option to `sdx wrap`. For example, we could build a x86 Linux-specific version of our `myapp` application by providing a x86 Linux tclkit.

```
c:\demo-dir> tclkit-cli sdx.kit wrap myapp.exe -vfs demo.vfs -runtime \
    tclkit-cli-runtime-linux
```

Here we assume `tclkit-cli-runtime-linux` is a tclkit application that has been built for our target Linux platform.

19.4.10. Starkit mount points

We need to touch upon one issue that you need to be aware of when using starkits and starpacks. As we discussed, starkits are seen by the Tcl I/O system as virtual file systems and as such have to be mounted. The mount point used for the VFS in starkits and starpacks is the local file system path for the starkit or starpack itself.

We can use our `demo` starkit for illustration purposes.

```
c:\demo-dir> tclkit-cli
% set kit [file normalize demo]
C:/demo-dir/demo
% file type $kit
file
```

As expected, `file type` returns the type of our starkit file as `file`. This is **before** we load it. However, **after** loading the starkit, we get a different result.

```
% source $kit
run_mode: sourced
% file type $kit
directory
```

It now show up as a `directory`! This is because the path `C:\demo-dir\demo` is now the mount point for the starkit and the root directory of the VFS.

We can confirm this with the `vfs::filesystem info` command.

```
% vfs::filesystem info
C:/demo-dir/demo C:/demo-dir/tclkit-cli.exe
```

Notice how `C:/demo-dir/demo` is listed as a file system. Furthermore, because tclkit applications themselves are structured as starpacks, our `tclkit-cli` executable file also shows up as a file system which leads to the following quirk.

```
% file type [info nameofexecutable]
directory
```

This is a consequence of the fact that tclkits and starpacks mount their contents as a file system at the mount point corresponding to their executable path. (At the risk of belaboring the point, we need to stress that this only applies to tclkit and starpack executables, not to the standard Tcl shells or applications.)

This implementation quirk is something you need to keep in mind when working with a starkit or starpack. For example, the following command

```
file copy [info nameofexecutable] target
```

will result in very different results in `tclsh` versus `tclkit-cli`. In the former case, `target` will be a copy of the `tclsh` executable. In the latter case, the `tclkit-cli` path is seen as a mounted directory and `target` will contain the entire directory structure contained within the tclkit VFS. You are encouraged to try the command and examine the difference.

19.4.11. Writable starkits

Our discussion so far has only involved read operations on starkits once they are constructed. Depending on the underlying technology used to implement a starkit, it is also possible to modify a starkit by writing to it. Among the three popular starkit technologies, Metakit, Vlerq and zip, only Metakit based starkits support writing.

To create a writable starkit, pass the `-writable` option to `sdx wrap`. For example,

```
c:\temp\demo>tclkit sdx.kit wrap demo.kit -writable
10 updates applied
```

You can then create, delete or write to files within the starkit using standard Tcl I/O commands **provided you are using a tclkit application based on Metakit technology**.

19.5. Chapter summary

In this chapter, we introduced virtual file systems and tclkits. Virtual file systems permit arbitrary structured data to be presented to the application as a local file system. It can then be accessed and worked on with the commonly used file and channel based commands. We saw examples of the utility of this for accessing remote files over FTP and for accessing process information as a file system.

We also saw the use of VFS for the purpose of creating single file Tcl applications. This has great benefits in terms of the ease with which applications can be deployed without the need for installers or additional packaging.

19.6. References

LAND2002
*Beyond TclKit - Starkits, Starpacks and other *stuff*, Landers, Proceedings of the 2002 Tcl Conference, http://www.digital-smarties.com/Tcl2002/tclkit.pdf