Sandbox IO

Aaron W. Hsu <arcfide@sacrideo.us>
Karissa R. McKelvey <krmckelv@indiana.edu>

2 March 2011

Contents

Library Definition: (mags sandboxes io)

This library contains various procedures for the construction and manipulation of a virtual filesystem for the sandbox.

Exports:

```
current-vfs                 peek-char                   flush-output-port
all-open-virtual-ports      write-char                  close-input-port
path-lookup                 newline                     close-output-port
max-vfs-space               open-input-file             close-port
current-vfs-space-use       open-output-file            display
run-sandboxes/io-tests      call-with-input-file        file-exists?
read                        call-with-output-file       delete-file
write                       with-input-from-file
read-char                   with-output-to-file
```

Imports:

```
    (srfi :64) (mags sandbox) (except (chezscheme) delete-file file-exists?  open-input-file
open-output-file call-with-input-file call-with-output-file with-input-from-file with-output-
to-file close-port)
```

1. Overview. This library creates a basic framework for a virtual filesystem for the sandbox. The advantage of this is to control and monitor space use as well as allow for extensive manipulation without threatening the integrity or structure of the original filesystem.

The `(current-vfs)`is initialized to (dir "/"), and can be changed by simply using it as a parameter. Specifics about the `(current-vfs)` an be found in the following section, titled "Virtual Filesystems". You can use any of the r6rs convenience I/O and filesystem operations in this library.

2. Nodes. Nodes are representative of either files or directories. Files, which are pairs containing a name and contents, look like this:

```
> (file "name" . "contents")
```

If the file has an *open* port, the contents will be that port. When you write to a port that is open in a node and then close it, the port is replaced by whatever was written. The `(current-vfs)` is the current virtual filesystem. Here is an example of writing to a node:

```
> (current-vfs (file "submission" . "Something")
> (define a (open-output-port "submission")
> a
<output-port submission>
> (current-vfs)
(file "submission" . <output port submission>)
> (write "Something" a)
> (close-output-port a)
> (current-vfs)
(file "submission" . "Something")
```

When we close the

Directories are simply nodes that contain a list of other nodes as their contents, for example:

```
(dir "/"
     (file "submission" . "This is a submission file")
     (file "foo" . "bar")
     (dir "cd"
  (file "name" . "contents")))
```

Here are some specific properties about nodes:

- Nodes are pairs.
- Nodes can represent either a file or directory.
- Both `file` nodes and `dir` nodes have an identifier at the `car`, either 'file or 'dir.

  Example: (file name . contents) and (dir (file name . contents))
- The `name` of a file or dir is a string representing the name.
- The `contents` of a node can be one of two things:

) **3. Virtual Filesystems.** Virtual filesystems assure that we can keep the top-level filesystem intact while performing operations.

`current-vfs` is a parameter that represents the current virtual filesystem that the sandbox is using. Each item in the list represents a `node`, which is essentially a path to the file which will be read into the sandbox. Specifics about nodes are explained in the above section.

⟨∗⟩ ≡
```
(define current-vfs
  (make-parameter
    '(dir "/")
    (lambda (x) (assert (or (null?  x) (list?  x))) x)))
```

4. Some basic test cases for the `current-vfs`

⟨Test current-vfs⟩ ≡
```
(test-begin "test-current-vfs")
(current-vfs
  '(dir "/"
        (file ("a" .  ""))
        (file ("b" .  ""))
        (dir "cd" (file ("e" .  "")) (file ("f" .  "")))))
(test-assert "current vfs is a node" (node?  (current-vfs)))
(test-equal
  "current vfs is what was initialized"
  '(dir "/"
        (file ("a" .  ""))
        (file ("b" .  ""))
        (dir "cd" (file ("e" .  "")) (file ("f" .  ""))))
  (current-vfs))
(test-end "test-current-vfs")
```

5. Retreiving Open Ports. When we create and continue to modify our virtual filesystem, we assume that we would want some way to retrieve all of the ports that are open in order to handle them accordingly.

`all-open-virtual-ports` is a thunk that returns a list of all currently open virtual ports in the virtual filesystem.

```
> (current-vfs)
(dir "/"
   (file "a" . "")
   (file "b" . "")
     (dir "cd"
        (file "e" . "")
        (file "f" . "")))
> (all-open-virtual-ports)
()

> (current-vfs
    (dir "/" (file "a" . <output-port a>)
```

```
                (file "b" . <output-port b>))
> (all-open-virtual-ports)
(<output-port a> <output-port b>)
```

Firstly, we must have added `nodes` to our `current-vfs`. As stated earlier, each of these `nodes` represents files or directories. To begin, `all-open-virtual-ports` pulls the nodes out of the `current-vfs` and loops over each node, collecting the open ports up into a list. It is implemented as such:

⟨Define all-open-virtual-ports⟩ ≡

```
(define (all-open-virtual-ports)
  (let dir-loop ([nodes (node-contents (current-vfs))]
                 [open-list '()]
                 [cd `(,(node-name (current-vfs)))])
    (if (null?  nodes)
        open-list
        (let ([node (car nodes)])
          (cond
            [(null?  node) open-list]
            [(file?  node)
             ⟨Handle File Node 6⟩]
            [(dir?  node)
             ⟨Handle Directory Node 7⟩]
            [else
             (errorf 'all-open-virtual-ports
               (format
                 "node ~s is not a valid filesystem node"
                 node))])))))
```

Exports: `all-open-virtual-ports`

6. Handling the nodes in `all-open-virtual-ports`.

For handling file nodes, if the file is open, we grab the filename from the file. We then put the filename, current-directory, and file-contents together.

After this, `cons` the result to the front of the current list that will be returned from `all-open-virtual-ports` once we are finished. Then we simply continue the loop over the `current-vfs` until we have completed the entire traverse of the list of nodes.

⟨Handle File Node⟩ ≡

```
(dir-loop
  (cdr nodes)
  (if (open?  node)
      (cons
        (let ([fname (node-name node)])
          (cons
            (path-list->path-string (cons fname cd))
            (node-contents node)))
        open-list)
      open-list)
  cd)
```

Captures: `dir-loop nodes open-list cd node`

7. For handling Directory Nodes, we simply perform a `dir-loop` over the nested directory, then call `append` on that finished list of open nodes to the list that will be returned later after walking down the `current-vfs`. We then continue the traverse.

⟨Handle Directory Node⟩ ≡

```
(dir-loop
  (cdr nodes)
  (append
    (dir-loop
      (node-contents node)
      '()
      (cons (node-name node) cd))
    open-list)
  cd)
```

Captures: `dir-loop nodes open-list cd node`

8. These are basic test cases for `all-open-virtual-ports`.
⟨Test all-open-virtual-ports⟩ ≡

```
(test-begin "all-open-virtual-ports")
(current-vfs
  '(dir "/"
        (file ("a" .  ""))
        (file ("b" .  ""))
        (dir "cd" (file ("e" .  "")) (file ("f" .  "")))))
(test-eq
  "all-open-virtual-ports with no open ports"
  '()
  (all-open-virtual-ports))
(test-end "all-open-virtual-ports")
```

9. Let's make sure that `all-open-virtual-ports` gets into the top-level.
⟨∗⟩ ≡

⟨Define all-open-virtual-ports 5⟩

10. Convenience Procedures. These procedures are used as convenience procedures in the implementation of nodes, either predicates or accessors for specific fields within the nodes.
⟨∗⟩ ≡

```
(define (file?  node)
  (and (pair?  node)
       (not (null?  node))
       (eq?  'file (car node))))
(define (dir?  node)
  (and (pair?  node) (not (null?  node)) (eq?  (car node) 'dir)))
(define (open?  file) (port?  (node-contents file)))
(define (node-name node) (and (node?  node) (cadr node)))
(define (node-contents node) (and (node?  node) (cddr node)))
(define (path-list->path-string path)
  (format "/~~a~^/~" path))
(define (string-null?  s) (zero?  (string-length s)))
(define (node?  node) (or (file?  node) (dir?  node)))
(define (node-cell node) (cdr node))
```

11. Here are some test cases for these procedures:
⟨Test helpers⟩ ≡

```
(test-begin "helpers")
(let ()
  (define file '(file "a" .  "abcd"))
  (define dir `(dir "~" ,file))
```

```
  (test-group "File nodes"
    (test-eq "a symbol is not a file" #f (file?  'a))
    (test-eq "a string is not a file" #f (file?  "a"))
    (test-eq "the empty list is not a file" #f (file?  '()))
    (test-assert "a file is a node" (node?  file))
    (test-assert "a file is a file" (file?  file))
    (test-equal
      "the file has the corrent node-name"
      "a"
      (node-name file))
    (test-equal
      "the file has the correct node-contents"
      "abcd"
      (node-contents file)))
  (test-group "Directory nodes"
    (test-eq "the empty list is not a directory" #f (dir?  '()))
    (test-eq "a symbol is not a directory" #f (dir?  'a))
    (test-eq "a string is not a directory" #f (dir?  "a"))
    (test-assert "the directory is a node" (node?  dir))
    (test-assert "the directory is a directory" (dir?  dir))
    (test-equal
      "the directory has the correct node-name"
      "~"
      (node-name dir))
    (test-equal
      "the directory has the correct node-contents"
      (list file)
      (node-contents dir)))
  (test-eq "the file is not open" #f (open?  file))
  (test-assert
    "string-null on the empty string"
    (string-null?  ""))
  (test-eq
    "string-null on the nonempty string"
    #f
    (string-null?  "abc")))
(test-end "helpers")
```

12. `path-lookup` takes one argument: a string representation of a path, and returns a node that represents that path as a node which can be either a file or directory or `#f` if a node can't be found with the given path.

```
> (path-lookup path) => node or #f

> (current-vfs)
(dir "/"
   (file "a" . "")
   (file "b" . "")
     (dir "cd"
        (file "e" . "")
        (file "f" . "")))

> (path-lookup "/a")
(file "a" . "")
```

```
> (path-lookup "/cd/e")
(file "e" . "")
```

⟨Define path-lookup⟩ ≡

```
(define (path-lookup path)
  (let loop ([path path] [node-list (list (current-vfs))])
    (let ([pcar (path-first path)])
      (if (string-null? pcar)
          (let ([pcdr (path-rest path)])
            ⟨Lookup node 14⟩)
          (let ([node ⟨Lookup node 14⟩])
            (and node (loop (path-rest path) (node-contents node))))))))
```

Exports: `path-lookup`

13. Let's make up some tests to verify the behavior of `path-lookup`.

⟨Test path-lookup⟩ ≡

```
(test-begin "path-lookup")
(current-vfs
  '(dir "/"
        (file "a" . "")
        (file "b" . "")
        (dir "cd" (file "e" . "") (file "f" . ""))))
(test-assert
  "path-lookup on file a"
  (file? (path-lookup "/a")))
(test-assert
  "path-lookup on file b"
  (file? (path-lookup "/b")))
(test-equal
  "path-lookup returns #f when it can't find a file"
  #f
  (path-lookup "g"))
(test-equal
  "path-lookup on dir e"
  '(file "e" . "")
  (path-lookup "/cd/e"))
(test-end "path-lookup")
```

14. Given the name of the file or directory we want to find and the list of nodes in the `current-vfs`, this procedure finds the `node` that represents that path and then returns it. It also takes a third argument which represents a predicate, or what we are assuming what type of node it should be, either a file or dir.

⟨Lookup node⟩ ≡

```
(define (correct-node? node)
  (and (node? node) (string=? name (node-name node))))
(let ([res (memp correct-node? node-list)])
  (and res (car res)))
```

Captures: `name node-list node?`

15. We now should throw the `path-lookup` procedure into the top-level.

⟨∗⟩ ≡

⟨Define path-lookup 12⟩

16. Virtual Ports. To support the virtual file system we must have a way to create ports that are directly linked to the virtual file system that we create. These are then used as the underlying framework for the above rewritten procedures. We support input and output text ports at the moment, but not input/output ports or binary ports. The basic idea is that when passed a file node, these procedures close over this node and create a port whose effects are visible in the node contents field.

```
(make-virtual-textual-input-port file-node)
(make-virtual-textual-output-port file-node)
```

Both of the above procedures return a port. They mutate the contents field of the `file-node`. It is important to note that while the port they return is open, that fact will be reflected in the contents field of a file node. Here is an example interaction that should illustrate this:

```
> (define file-node (cons* 'file "test_file" "Something"))
> file-node
(file "test_file" . "Something")
> (define op (make-virtual-textual-output-port file-node))
> op
#<text output port>
> file-node
(file "test_file" . #<text output port>)
> (put-string op "This isn't something.
")
> file-node
(file "test_file" . #<text output port>)
> (close-port op)
> file-node
(file "test_file" . "This isn't something.
")
>
```

Note that the actual effects of the writing are not visible in the node itself until that output port has been closed. This actually means that we are enforcing a single access restriction on the file nodes.

17. Protecting from the overuse of space. When writing to a `virtual-textual-output-port`, we need to make sure that we use a reasonable amount of space. In this sense, we must keep some type of record of how much space we have written so far to the port, in this case it is a parameter `current-vfs-space`. We also need a parameter which can be changed, that is, the max amount of space we wish to write, `max-vfs-space`.

⟨∗⟩ ≡

```
(define current-vfs-space-use
  (make-parameter 0 (lambda (x) (assert (integer?  x)) x)))
(define max-vfs-space
  (make-parameter 300 (lambda (x) (assert (integer?  x)) x)))
```

18. Rewriting Convenience I/O. We can use these procedures to read from and write to files. This section overrides the (chezscheme) Convenience I/O to utilize the virtual filesystem.

The new `open-input-file` and `open-output-file` take a path, and transform it into the respective node representation. The node must be already present in the `current-vfs`, or the procedure throws an error. Then, they open an input or output port by calling `make-virtual-input-port` on the node that was found from `path-lookup`.

⟨∗⟩ ≡

```
(define (open-input-file path)
  (let ([node (path-lookup path)])
    (if (not node)
```

```
        (errorf 'open-input-file
          "Path ~s not found in the current-vfs"
          path)
        (make-virtual-textual-input-port node))))
(define (open-output-file path)
  (let ([node (path-lookup path)])
    (if (not (node?  node))
        (errorf 'open-output-file
          "Path ~s not found in the current-vfs"
          path)
        (make-virtual-textual-output-port node))))
(define (call-with-input-file filename proc)
  (let ([p (open-input-file filename)])
    (let-values ([v* (proc p)])
      (close-port p)
      (apply values v*))))
(define (call-with-output-file filename proc)
  (let ([p (open-output-file filename)])
    (let-values ([v* (proc p)])
      (close-port p)
      (apply values v*))))
(define (with-input-from-file path thunk)
  (parameterize ([current-input-port (open-input-file path)])
    (thunk)))
(define (with-output-to-file path thunk)
  (parameterize ([current-output-port
                  (open-output-file path)])
    (thunk)))
(define (close-port port)
  (when (input-port?  port) (close-input-port port))
  (when (output-port?  port) (close-output-port port)))
```

19. Filesystem operations

⟨∗⟩ ≡

```
(define (delete-file path)
  (let ([new-vfs (let loop ([path path]
                            [node-list (list (current-vfs))])
                   (let ([pcar (path-first path)])
                     (if (string-null?  pcar)
                         (let ([pcdr (path-rest path)])
                           (remove
                             ⟨Lookup node 14⟩
                             node-list))
                         (let ([node ⟨Lookup node 14⟩])
                           (and node
                                (append
                                  (remp dir?  (current-vfs))
                                  (loop
                                    (path-rest path)
                                    (node-contents node)))))))))])
    (and new-vfs (current-vfs new-vfs))))
(define (file-exists?  path) (and (path-lookup path)))
```

20. Following are test cases for the filesystem operations
⟨Test filesystem operations⟩ ≡

```
(test-group
  "filesystem operations"
  (parameterize ([current-vfs
                  '(dir "/"
                        (file "a" .  "")
                        (file "b" .  "")
                        (dir "cd" (file "e" .  "") (file "f" .  "")))])
    (test-assert
      "delete-file on top-level file"
      (delete-file "/a"))
    (test-equal
      "current-vfs is correct1"
      '(dir "/"
            (file "b" .  "")
            (dir "cd" (file "e" .  "") (file "f" .  "")))
      (current-vfs))
    (test-assert
      "delete-file on nested file"
      (delete-file "/cd/e"))
    (test-equal
      "current-vfs is correct2"
      '(dir "/" (file "b" .  "") (dir "cd" (file "f" .  "")))
      (current-vfs))
    (test-equal
      "current-vfs is preserved"
      '(dir "/" (file "b" .  "") (dir "cd" (file "f" .  "")))
      (current-vfs))
    (parameterize ([current-vfs
                    '(dir "/" (file "b" .  "") (dir "cd" (file "f" .  "")))])
      (test-assert "file-exists1" (file-exists?  "/b"))
      (test-equal
        "file-exists when file doesn't exist"
        #f
        (file-exists?  "/asdfasdf"))
      (test-assert "file exists2" (file-exists?  "/cd/f")))))
```

21. Input Ports: Implementation. Virtual textual input ports are based on custom input ports. We are using the R6RS version of custom input ports. These ports require that we have five values:

`id` Just some identifier for the port. We are using the file name of the node.
`r!` This is the actual reader procedure that gives the system back the data it needs.
`gp` Is the procedure that returns the position of the port.
`sp!` This sets the position of the port.
`close` This does any final closing actions that need to be performed.

The `make-virtual-textual-input-port` procedure will create a custom port from the node, with the file node's name as the `id` and we will use close to restore the file node. During the actual opening of the file node, we mutate the node cell so that its contents reflects that we have opened the file. In this case, we replace the contents with the port that we have just created.
⟨Define make-virtual-textual-input-port⟩ ≡

```
(define (make-virtual-textual-input-port node)
  (assert (file?  node))
```

```
  (let ([file-name (node-name node)]
        [file-contents (node-contents node)]
        [contents-cell (node-cell node)]
        [port-pos 0])
    (when (port?  file-contents)
      (errorf file-name "The file is already open."))
    (assert (string?  file-contents))
    (let ([ip (make-custom-textual-input-port file-name
                ⟨Virtual input port reader 22⟩
                (lambda () port-pos) (lambda (pos) (set!  port-pos pos))
                (lambda () (set-cdr!  contents-cell file-contents)))])
      (set-cdr!  contents-cell ip)
      ip)))
```

Exports: `make-virtual-textual-input-port`

22. The virtual input port reader maintains a current pointer into the `file-contents` string, and moves through it, maintaining that port position. The reader has the following signature:

```
(r! string start n) => count
```

The reader should fill the string starting at `start` and filling in at most `n` characters. These characters should be filled from the contents of the file. The count returned are the actual number of characters thrown into the string.

⟨Virtual input port reader⟩ ≡

```
(define file-length (string-length file-contents))
(lambda (string start n)
  (define end (+ start n))
  (let loop ([i start] [count 0] [port-i port-pos])
    (cond
      [(>= port-i file-length) (set!  port-pos port-i) count]
      [(>= i end) (set!  port-pos port-i) count]
      [else
       (string-set!  string i (string-ref file-contents port-i))
       (loop (fx1+ i) (fx1+ count) (fx1+ port-i))])))
```

Captures: `file-contents port-pos`

23. Now we will throw the definition up to the top-level.

⟨∗⟩ ≡

⟨Define make-virtual-textual-input-port 21⟩

24. Output Ports: Implementation. The output ports for this library are implemented in a similar way as the input ports, found in the above section.

`make-virtual-output-port` takes a node and opens an output port on the virtual filesystem. It takes one argument which must be a file node.

⟨Define make-virtual-textual-output-port⟩ ≡

```
(define (make-virtual-textual-output-port node)
  (assert (file?  node))
  (let ([file-name (node-name node)]
        [file-contents (node-contents node)]
        [contents-cell (node-cell node)]
        [contents '()])
    (when (port?  file-contents)
      (errorf file-name "The file is already open."))
```

```
(assert (string?  file-contents))
(current-vfs-space-use
  (- (current-vfs-space-use) (string-length file-contents)))
(let ([op (make-custom-textual-output-port file-name
            ⟨Virtual output port writer 25⟩
            (lambda () #f) (lambda (pos) #f)
            ⟨Virtual output port closer 26⟩)])
  (set-cdr!  contents-cell op)
  op)))
```

Exports: `make-virtual-textual-output-port`

25. The vrtual output port writer maintains a current pointer into the `file-contents` string, and writes to it, maintaining that port position.

The writer writes up to `n` characters from a string and returns an integer representation of the number of characters written. It then records the total number of characters in the `current-vfs-space-use` in order to ensure the protection of space.

⟨Virtual output port writer⟩ ≡

```
(lambda (string start n)
  (let ([s (substring string start (fx+ start n))]
        [usable-space (- (max-vfs-space) (current-vfs-space-use))])
    (when (< usable-space n)
      (errorf 'max-vfs-space
        (format
          "The value to be written, ~s surpases the maximum allotted space."
          s)))
    (set!  contents (cons s contents))
    (current-vfs-space-use (+ (current-vfs-space-use) n))
    (string-length s)))
```

Captures: `contents`

26.

⟨Virtual output port closer⟩ ≡

```
(lambda ()
  (set-cdr!
    contents-cell
    (apply string-append (reverse contents))))
```

Captures: `contents contents-cell`

27.

⟨∗⟩ ≡

⟨Define make-virtual-textual-output-port 24⟩

28. Here are some test cases for the virtual ports:

⟨Test virtual ports⟩ ≡

```
(test-begin "Virtual Ports")
(let ([file-node (cons* 'file "test" "Something.\n")])
  (let ([ip (make-virtual-textual-input-port file-node)])
    (test-eq "read on the input port" 'Something.  (read ip))
    (close-port ip))
  (let ([op (make-virtual-textual-output-port file-node)])
    (put-string op "Nothing.\n")
```

```
    (close-port op))
  (test-equal
    "write on the file-node"
    "Nothing.\n"
    (node-contents file-node)))
(test-begin "vfs space")
(let ()
  (current-vfs-space-use 0)
  (test-equal
    "current-vfs-space-use after write"
    0
    (current-vfs-space-use))
  (test-equal "default max-vfs-space" 300 (max-vfs-space))
  (current-vfs-space-use 3)
  (max-vfs-space 5)
  (test-equal
    "after changing max-vfs-space"
    5
    (max-vfs-space))
  (test-equal
    "after changing current-vfs-space-use"
    3
    (current-vfs-space-use)))
(test-end "vfs space")
(max-vfs-space 300)
(test-end "Virtual Ports")
```

29. Sandbox I/O Tests. We use testing suite SRFI 64, which allows us to easily write test cases for each of our procedures. Simply run `(test-all)` to run all of the tests for these procedures.

⟨∗⟩ ≡

```
(define (run-sandboxes/io-tests)
  (parameterize ([test-runner-current (test-runner-simple)])
    ⟨Test current-vfs 4⟩
    ⟨Test all-open-virtual-ports 8⟩
    ⟨Test helpers 11⟩
    ⟨Test path-lookup 13⟩
    ⟨Test virtual ports 28⟩
    ⟨Test filesystem operations 20⟩))
```

This concludes the definition of (mags sandboxes io).