Scsh: The Reference Manual

Version 8.14

February 26, 2024

Contents

1 Introduction

This is the reference manual for `scsh`, a Unix shell that is embedded within Scheme. Scsh is a Scheme designed for writing useful standalone Unix programs and shell scripts—it spans a wide range of application, from "script" applications usually handled with `sh`, to more standard systems applications usually written in C.

Scsh is built as a library on top of scheme48, and has two components: a process notation for running programs and setting up pipelines and redirections, and a syscall library for low-level access to the operating system.

This manual gives a complete description of scsh. A general discussion of the design principles behind scsh can be found in a companion paper "A Scheme Shell" (reference).

1.1 Copyright & License

Scsh is open source. The complete source comes with the standard distribution. Scsh has an ideologically hip, BSD-style license.

We note that the code is a rich source for other Scheme implementations to mine. Not only the *code*, but the *APIs* are available for implementors working on Scheme environments for systems programming. These APIs represent years of work, and should provide a big head-start on any related effort. (Just don't call it "scsh," unless it's *exactly* compliant with the scsh interfaces.)

Take all the code you like; we'll just write more.

1.2 Obtaining Scsh

The current version of scsh is still in development. We're using git as a source code management system, and the primary repository is hosted on github at scheme/scsh.

This section will be updated to point at the final distribution once the next version is released.

2 Process Notation

Scsh has a notation for controlling Unix processes that takes the form of s-expressions; this notation can then be embedded inside of standard Scheme code. The basic elements of this notation are *process forms*, *extended process forms*, and *redirections*.

2.1 Extended Process Forms and I/O Redirections

An *extended process form* is a specification of a Unix process to run, in a particular I/O environment:

```
epf ::= (pf redir-1 ... redir-n)
```

where `pf` is a process form and the `redir`s are redirection specs.

A *redirection spec* is one of:

`(< [fdes] file-name)`	Open file for read.
`(> [fdes] file-name)`	Open file create/truncate.
`(<< [fdes] object)`	Use `object`'s printed rep.
`(>> [fdes] file-name)`	Open file for append.
`(= [fdes] fdes/port)`	Dup2
`(- fdes/port)`	Close `fdes/port`
`stdports`	0,1,2 dup'd from standard ports.

The input redirections default to file descriptor 0; the output redirections default to file descriptor 1.

The subforms of a redirection are implicitly backquoted, and symbols stand for their print-names. So `(> ,x)` means "output to the file named by Scheme variable `x`," and `(< /usr/shivers/.login)` means "read from `/usr/shivers/.login`."

Here are two more examples of I/O redirection:

```
(< ,(vector-ref fv i))
(>> 2 /tmp/buf)
```

These two redirections cause the file `fv[i]` to be opened on stdin, and `/tmp/buf` to be opened for append writes on stderr.

The redirection `(<< object)` causes input to come from the printed representation of `object`. For example,

```
(<< "The quick brown fox jumped over the lazy dog.")
```

causes reads from stdin to produce the characters of the above string. The object is converted to its printed representation using the `display` procedure, so

```
(<< (A five element list))
```

is the same as

```
(<< "(A five element list)")
```

is the same as

```
(<< ,(reverse '(list element five A)))
```

(Here we use the implicit backquoting feature to compute the list to be printed.)

The redirection `(= fdes fdes/port)` causes `fdes/port` to be dup'd into file descriptor `fdes`. For example, the redirection

```
(= 2 1)
```

causes stderr to be the same as stdout. `fdes/port` can also be a port, for example:

```
(= 2 ,(current-output-port))
```

causes stderr to be dup'd from the current output port. In this case, it is an error if the port is not a file port (e.g., a string port).

More complex redirections can be accomplished using the `begin` process form, discussed below, which gives the programmer full control of I/O redirection from Scheme.

2.1.1   Port and File Descriptor Sync

It's important to remember that rebinding Scheme's current I/O ports (e.g., using `call-with-input-file` to rebind the value of `(current-input-port)`) does *not* automatically "rebind" the file referenced by the Unix stdio file descriptors 0, 1, and 2. This is impossible to do in general, since some Scheme ports are not representable as Unix file descriptors. For example, many Scheme implementations provide "string ports," that is, ports that collect characters sent to them into memory buffers. The accumulated string can later be retrieved from the port as a string. If a user were to bind `(current-output-port)` to such a port, it would be impossible to associate file descriptor 1 with this port, as it cannot be represented in Unix. So, if the user subsequently forked off some other program as a subprocess, that program would of course not see the Scheme string port as its standard output.

To keep stdio synced with the values of Scheme's current I/O ports, use the special redirection `stdports`. This causes 0, 1, 2 to be redirected from the current Scheme standard ports. It is equivalent to the three redirections:

```
(= 0 ,(current-input-port))
(= 1 ,(current-output-port))
(= 2 ,(error-output-port))
```

The redirections are done in the indicated order. This will cause an error if one of the current I/O ports isn't a Unix port (e.g., if one is a string port). This Scheme/Unix I/O synchronisation can also be had in Scheme code (as opposed to a redirection spec) with the `(stdports->stdio)` procedure.

2.2 Process Forms

A *process form* specifies a computation to perform as an independent Unix process. It can be one of the following:

```
(begin . scheme-code)            ; Run scheme-code in a fork.
(| pf-1 ... pf-n) ; Simple pipeline (|+ connect-list pf-1 ... pf-
n)  ; Complex pipeline
(epf . epf)                      ; An extended process form.
(prog arg-1 ... arg-n)           ; Default: exec the program.
```

The default case `(prog arg-1 ... arg-n)` is also implicitly backquoted. That is, it is equivalent to:

```
(begin (apply exec-path `(prog arg-1 ... arg-n)))
```

`exec-path` is the version of the `exec()` system call that uses scsh's path list to search for an executable. The program and the arguments must be either strings, symbols, or integers. Symbols and integers are coerced to strings. A symbol's print-name is used. Integers are converted to strings in base 10. Using symbols instead of strings is convenient, since it suppresses the clutter of the surrounding `"..."` quotation marks. To aid this purpose, scsh reads symbols in a case-sensitive manner, so that you can say

```
(more Readme)
```

and get the right file.

A `connect-list` is a specification of how two processes are to be wired together by pipes. It has the form `((from-1 from-2 ... to) ...)` and is implicitly backquoted. For example,

```
(\|+ ((1 2 0) (3 1)) pf-1 pf-2)
```

runs `pf-1` and `pf-2`. The first clause `(1 2 0)` causes `pf-1`'s stdout (1) and stderr (2) to be connected via pipe to `pf-2`'s stdin (0). The second clause `(3 1)` causes `pf-1`'s file descriptor 3 to be connected to `pf-2`'s file descriptor 1.

The `begin` process form does a `stdio->stdports` synchronisation in the child process before executing the body of the form. This guarantees that the `begin` form, like all other process forms, "sees" the effects of any associated I/O redirections.

Note that RnRS does not specify whether or not | and |+ are readable symbols. Scsh does.

2.3   Using Extended Process Forms in Scheme

Process forms and extended process forms are *not* Scheme. They are a different notation for expressing computation that, like Scheme, is based upon s-expressions. Extended process forms are used in Scheme programs by embedding them inside special Scheme forms. There are three basic Scheme forms that use extended process forms: `exec-epf`, `&`, and `run`.

```
(exec-epf . epf) ; => no return value (syntax)
(& . epf)        ; => proc (syntax)
(run . epf)      ; => status (syntax)
```

The `(exec-epf . epf)` form nukes the current process: it establishes the I/O redirections and then overlays the current process with the requested computation.

The `(& . epf)` form is similar, except that the process is forked off in background. The form returns the subprocess' process object.

The `(run . epf)` form runs the process in foreground: after forking off the computation, it waits for the subprocess to exit, and returns its exit status.

These special forms are macros that expand into the equivalent series of system calls. The definition of the `exec-epf` macro is non-trivial, as it produces the code to handle I/O redirections and set up pipelines.

However, the definitions of the `&` and `run` macros are very simple:

```
(& . epf)   (fork (lambda () (exec-epf . epf)))
(run . epf) (wait (& . epf))
```

2.3.1 Procedures and Special Forms

It is a general design principle in scsh that all functionality made available through special syntax is also available in a straightforward procedural form. So there are procedural equivalents for all of the process notation. In this way, the programmer is not restricted by the particular details of the syntax.

Here are some of the syntax/procedure equivalents:

```
|           fork/pipe
|+          fork/pipe+
exec-epf exec-path
redirection open, dup
&           fork
run         exec + fork
```

Having a solid procedural foundation also allows for general notational experimentation using Scheme's macros. For example, the programmer can build his own pipeline notation on top of the `fork` and `fork/pipe` procedures.

gives the full story on all the procedures in the syscall library.

2.3.2 Interfacing Process Output to Scheme

There is a family of procedures and special forms that can be used to capture the output of processes as Scheme data. These forms all fork off subprocesses, collecting the process' output to stdout in some form or another. The subprocess runs with file descriptor 1 and the current output port bound to a pipe. Furthermore, each of these forms is a simple expansion into calls to analogous procedures. For example, `(run/port . epf)` expands into `(run/port* (lambda () (exec-epf . epf)))`.

```
(run/port epf)
(run/port* thunk) → port?
  thunk : (-> any/c)
```

Returns a port open on process's stdout. Returns immediately after forking child process.

```
(run/file epf)
(run/file* thunk) → string?
  thunk : (-> any/c)
```

Returns the name of a temp file containing the process's output. Returns when the process exits.

```
(run/string epf)
```

9

```
(run/string* thunk) → string?
  thunk : (-> any/c)
```

Returns a string containing the process' output. Returns when an eof is read.

```
(run/strings epf)
(run/strings* thunk) → (listof string?)
  thunk : (-> any/c)
```

Splits process' output into a list of newline-delimited strings. The delimiting newlines are not part of the returned strings. Returns when an eof is read.

```
(run/sexp epf)
(run/sexp* thunk) → any/c
  thunk : (-> any/c)
```

Returns a single object from process' stdout with `read`. Returns as soon as the read completes.

```
(run/sexps epf)
(run/sexps* thunk) → (listof any/c)
  thunk : (-> any/c)
```

Repeatedly reads objects from process' stdout with `read`. Returns accumulated list upon eof.

Parsing Input from Ports

The following procedures are also of utility for generally parsing input streams in scsh:

```
(port->string port) → string?
  port : port?
```

Reads the port until eof, then returns the accumulated string.

```
(port->sexp-list port) → (listof any/c)
  port : port?
```

Repeatedly reads data from the port until eof, then returns the accumulated list of items.

```
(port->string-list port) → (listof string?)
  port : port?
```

Repeatedly reads newline-terminated strings from the port until eof, then returns the accumulated list of strings. The delimiting newlines are not part of the returned strings.

```
(port->list reader port) → (listof any/c)
  reader : (-> port? any/c)
  port : port?
```

Generalises `port->sexp-list` and `port->string-list`. Uses `reader` to repeatedly read objects from a port and accumulates these objects into a list, which is returned upon eof. `port->sexp-list` and `port->string-list` are trivial to define, being merely `port->list` curried with the appropriate parsers:

```
(port->string-list port) => (port->list read-line port)
(port->sexp-list   port) => (port->list read port)
```

The following compositions also hold:

```
run/string* => (compose port->string run/port*)
run/strings* => (compose port->string-list run/port*)
run/sexp* => (compose read run/port*)
run/sexps* => (compose port->sexp-list run/port*)
```

```
(port-fold port reader op seeds ...+) → any/c ...+
  port : port?
  reader : (-> port? any/c)
  op : (-> any/c any/c ...+ (values any/c ...+))
  seeds : any/c
```

This procedure can be used to perform a variety of iterative operations over an input stream. It repeatedly uses `reader` to read an object from `port`. If the first read returns eof, then the entire `port-fold` operation returns the seeds as multiple values.

If the first read operation returns some other value `v`, then `op` is applied to `v` and the seeds: `(op v . seeds)`. This should return a new set of seed values, and the reduction then loops, reading a new value from the port, and so forth. If multiple seed values are used, then `op` must return multiple values.

For example,

```
(port->list reader port)
```

could be defined as

```
(reverse (port-fold port reader cons '()))
```

An imperative way to look at `port-fold` is to say that it abstracts the idea of a loop over a stream of values read from some port, where the seed values express the loop state.

2.4 More Complex Process Operations

The procedures and special forms in §2.3 "Using Extended Process Forms in Scheme" provide for the common case, where the programmer is only interested in the output of the process. These special forms and procedures provide more complicated facilities for manipulating processes.

2.4.1 Pids and Ports Together

```
(run/port+proc epf)
(run/port+proc* thunk) → [port port?] [proc process?]
  thunk : (-> any/c)
```

This special form and its analogous procedure can be used if the programmer also wishes access to the process' pid, exit status, or other information. They both fork off a subprocess, returning two values: a port open on the process' stdout (and current output port), and the subprocess's process object. A process object encapsulates the subprocess' process id and exit code; it is the value passed to the `wait` system call.

For example, to uncompress a tech report, reading the uncompressed data into scsh, and also be able to track the exit status of the decompression process, use the following:

```
(receive (port child) (run/port+proc (zcat tr91-145.tex.Z))
  (let* ((paper (port->string port))
         (status (wait child)))
    ... use paper, status, and child here ...))
```

Note that you must *first* do the `port->string` and *then* do the wait—the other way around may lock up when the zcat fills up its output pipe buffer.

2.4.2 Multiple Stream Capture

Occasionally, the programmer may want to capture multiple distinct output streams from a process. For instance, he may wish to read the stdout and stderr streams into two distinct strings. This is accomplished with the `run/collecting` form and its analogous procedure, `run/collecting*`.

```
(run/collecting fds ...+ epf)
(run/collecting* fds thunk)
 → [status integer?] [port port?] ...+
  fds : (listof integer?)
  thunk : (-> any/c)
```

These guys run processes that produce multiple output streams and return ports open on these streams. To avoid issues of deadlock, `run/collecting` doesn't use pipes. Instead, it first runs the process with output to temp files, then returns ports open on the temp files. For example,

```
(run/collecting (1 2) (ls))
```

runs `ls` with stdout (fd 1) and stderr (fd 2) redirected to temporary files. When the `ls` is done, `run/collecting` returns three values: the `ls` process' exit status, and two ports open on the temporary files. The files are deleted before `run/collecting` returns, so when the ports are closed, they vanish. The `fds` list of file descriptors is implicitly backquoted by the special-form version.

For example, if Kaiming has his mailbox protected, then

```
(receive (status out err)
         (run/collecting (1 2) (cat /usr/kmshea/mbox))
  (list status (port->string out) (port->string err)))
```

might produce the list

```
(256 "" "cat: /usr/kmshea/mbox: Permission denied")
```

What is the deadlock hazard that causes `run/collecting` to use temp files? Processes with multiple output streams can lock up if they use pipes to communicate with Scheme I/O readers. For example, suppose some Unix program `myprog` does the following:

1. First, outputs a single "(" to stderr.

2. Then, outputs a megabyte of data to stdout.

3. Finally, outputs a single ")" to stderr, and exits.

Our scsh programmer decides to run `myprog` with stdout and stderr redirect *via Unix pipes* to the ports `port1` and code{port2}, respectively. He gets into trouble when he subsequently says `(read port2)`. The Scheme `read` routine reads the open paren, and then hangs in a read()} system call trying to read a matching close paren. But before `myprog` sends the close paren down the stderr pipe, it first tries to write a megabyte of data to the stdout pipe. However, Scheme is not reading that pipe—it's stuck waiting for input on stderr. So the stdout pipe quickly fills up, and `myprog` hangs, waiting for the pipe to drain. The `myprog` child is stuck in a stdout/`port1` write; the Scheme parent is stuck in a stderr/`port2` read. Deadlock.

Here's a concrete example that does exactly the above:

```
(receive (status port1 port2)
             (run/collecting (1 2)
         (begin
            ;; Write an open paren to stderr.
            (run (echo "(") (= 1 2))
            ;; Copy a lot of stuff to stdout.
            (run (cat /usr/dict/words))
            ;; Write a close paren to stderr.
            (run (echo ")") (= 1 2))))

 ;; OK. Here, I have a port PORT1 built over a pipe
 ;; connected to the BEGIN subproc's stdout, and
 ;; PORT2 built over a pipe connected to the BEGIN
 ;; subproc's stderr.
 (read port2) ; Should return the empty list.
 (port->string port1)) ; Should return a big string.
```

In order to avoid this problem, `run/collecting` and `run/collecting*` first run the child process to completion, buffering all the output streams in temp files (using the `temp-file-channel` procedure). When the child process exits, ports open on the buffered output are returned. This approach has two disadvantages over using pipes:

- The total output from the child output is temporarily written to the disk before returning from `run/collecting`. If this output is some large intermediate result, the disk could fill up.

- The child producer and Scheme consumer are serialised; there is no concurrency overlap in their execution.

However, it remains a simple solution that avoids deadlock. More sophisticated solutions can easily be programmed up as needed—`run/collecting*` itself is only 12 lines of simple code.

See `temp-file-channel` for more information on creating temp files as communication channels.

2.5   Conditional Process Sequencing Forms

These forms allow conditional execution of a sequence of processes.

```
(|| pf ...+)
```

Run each proc until one completes successfully (i.e., exit status zero). Return true if some proc completes successfully; otherwise `#f`.

```
(&& pf ...+)
```

Run each proc until one fails (i.e., exit status non-zero). Return true if all procs complete successfully; otherwise #f.

2.6 Process Filters

These procedures are useful for forking off processes to filter text streams.

```
(make-char-port-filter filter) → procedure?
  filter : (-> character? character?)
```

Returns a procedure that when called, repeatedly reads a character from the current input port, applies filter to the character, and writes the result to the current output port. The procedure returns upon reaching eof on the input port.

For example, to downcase a stream of text in a spell-checking pipeline, instead of using the Unix tr A-Z a-z command, we can say:

```
(run (\| (delatex)
         (begin ((char-filter char-downcase))) ; tr A-Z a-z
         (spell)
         (sort)
         (uniq))
     (< scsh.tex)
     (> spell-errors.txt))
```

```
(make-string-port-filter filter [buflen]) → procedure?
  filter : (-> string? string?)
  buflen : integer? = 1024
```

Returns a procedure that when called, repeatedly reads a string from the current input port, applies filter to the string, and writes the result to the current output port. The procedure returns upon reaching eof on the input port.

The optional buflen argument controls the number of characters each internal read operation requests; this means that filter will never be applied to a string longer than buflen chars.

3 System Calls

Scsh provides (almost) complete access to the basic Unix kernel services: processes, files, signals and so forth. These procedures comprise a Scheme binding for Posix, with a few of the more standard extensions thrown in (e.g., symbolic links, `fchown`, `fstat`, sockets).

3.1 Errors

Scsh syscalls never return error codes, and do not use a global `errno` variable to report errors. Errors are consistently reported by raising exceptions. This frees up the procedures to return useful values, and allows the programmer to assume that *if a syscall returns, it succeeded.* This greatly simplifies the flow of the code from the programmer's point of view.

Since Scheme does not yet have a standard exception system, the scsh definition remains somewhat vague on the actual form of exceptions and exception handlers. When a standard exception system is defined, scsh will move to it. For now, scsh uses the scm exception system, with a simple sugaring on top to hide the details in the common case.

System call error exceptions contain the Unix `errno` code reported by the system call. Unlike C, the `errno` value is a part of the exception packet, it is *not* accessed through a global variable.

For reference purposes, the Unix `errno` numbers are bound to the variables `errno/perm`, `errno/noent`, etc. System calls never return \error/intr—they automatically retry.

```
(errno-error errno syscall data ...) → any
  errno : integer?
  syscall : string?
  data : any/c
```

Raises a Unix error exception for Unix error number *errno*. The *syscall* and *data* arguments are packaged up in the exception packet passed to the exception handler.

```
(with-errno-handler handler-spec ... body)
(with-errno-handler* handler thunk) → any/c ...
  handler : (-> integer? (listof any/c) (values any/c ...))
  thunk : (-> (values any/c ...))
```

Unix syscalls raise error exceptions by calling `errno-error`. Programs can use `with-errno-handler*` to establish handlers for these exceptions.

If a Unix error arises while *thunk* is executing, *handler* is called on two arguments like this:

```
(handler errno packet)
```

packet is a list of the form

```
(errno-msg syscall data ...)
```

where *errno-msg* is the standard Unix error message for the error, *syscall* is the procedure that generated the error, and *data* is a list of information generated by the error, which varies from syscall to syscall.

If *handler* returns, the handler search continues upwards. *handler* can acquire the exception by invoking a saved continuation. This procedure can be sugared over with the following syntax:

```
(with-errno-handler
      ((errno packet) clause ...)
   body ...+)
```

This form executes the body forms with a particular errno handler installed. When an errno error is raised, the handler search machinery will bind variable *errno* to the error's integer code, and variable *packet* to the error's auxiliary data packet. Then, the clauses will be checked for a match. The first clause that matches is executed, and its value is the value of the entire `with-errno-handler` form. If no clause matches, the handler search continues.

Error clauses have two forms

```
((errno ...) body ...)
(else body ...)
```

In the first type of clause, the *errno* forms are integer expressions. They are evaluated and compared to the error's errno value. An `else` clause matches any errno value. Note that the *errno* and *data* variables are lexically visible to the error clauses.

Example:

```
(with-errno-handler
     ((errno packet) ; Only handle 3 particular errors.
      ((errno/wouldblock errno/again)
        (loop))
      ((errno/acces)
        (format #t "Not allowed access!")
        #f))

  (foo frobbotz)
  (blatz garglemumph))
```

It is not defined what dynamic context the handler executes in, so that fluid variables cannot reliably be referenced.

Note that Scsh system calls always retry when interrupted, so that the `errno/intr` exception is never raised. If the programmer wishes to abort a system call on an interrupt, he should have the interrupt handler explicitly raise an exception or invoke a stored continuation to throw out of the system call.

3.1.1 Interactive Mode and Error Handling

Scsh runs in two modes: interactive and script mode. It starts up in interactive mode if the scsh interpreter is started up with no script argument. Otherwise, scsh starts up in script mode. The mode determines whether scsh prints prompts in between reading and evaluating forms, and it affects the default error handler. In interactive mode, the default error handler will report the error, and generate an interactive breakpoint so that the user can interact with the system to examine, fix, or dismiss from the error. In script mode, the default error handler causes the scsh process to exit.

When scsh forks a child with `(fork)`, the child resets to script mode. This can be overridden if the programmer wishes.

3.2 I/O

3.2.1 Standard RnRS I/O Procedures

In scsh, most standard RnRS I/O operations (such as `display` or `read-char`) work on both integer file descriptors and Scheme ports. When doing I/O with a file descriptor, the I/O operation is done directly on the file, bypassing any buffered data that may have accumulated in an associated port. Note that character-at-a-time operations such as `read-char` are likely to be quite slow when performed directly upon file descriptors.

The standard RnRS procedures `read-char`, `char-ready?`, `write`, `display`, `newline`, and `write-char` are all generic, accepting integer file descriptor arguments as well as ports. Scsh also mandates the availability of `format`, and further requires `format` to accept file descriptor arguments as well as ports.

The procedures `peek-char` and `read` do *not* accept file descriptor arguments, since these functions require the ability to read ahead in the input stream, a feature not supported by Unix I/O.

3.2.2 Port Manipulation and Standard Ports

```
(close-after port consumer) → any
  port : port?
  consumer : (-> port? any)
```

Returns `(consumer port)`, but closes the port on return. No dynamic-wind magic.

```
(error-output-port) → port?
```

This procedure is analogous to `current-output-port`, but produces a port used for error messages—the scsh equivalent of stderr.

```
(with-current-input-port* port thunk) → any
  port : port?
  thunk : (-> any)
(with-current-output-port* port thunk) → any
  port : port?
  thunk : (-> any)
(with-error-output-port* port thunk) → any
  port : port?
  thunk : (-> any)
```

These procedures install *port* as the current input, current output, and error output port, respectively, for the duration of a call to *thunk* and return *thunk*'s value(s).

```
(with-current-input-port port body ...+)
(with-current-output-port port body ...+)
(with-error-output-port port body ...+)
```

These special forms are simply syntactic sugar for the `with-current-input-port*` procedure and friends.

```
(close fd/port) → boolean?
  fd/port : (or/c integer? port?)
```

Closes the port or file descriptor.

If *fd/port* is a file descriptor, and it has a port allocated to it, the port is shifted to a new file descriptor created with `(dup fd/port)` before closing `fd/port`. The port then has its revealed count set to zero. This reflects the design criteria that ports are not associated with file descriptors, but with the streams they denote.

To close a file descriptor, and any associated port it might have, you must instead say one of (as appropriate):

```
(close (fdes->inport  fd))
(close (fdes->outport fd))
```

The procedure returns true if it closed an open port. If the port was already closed, it returns false; this is not an error.

```
(stdports->stdio) → undefined
```

Synchronises Unix' standard I/O file descriptors and Scheme's current I/O ports. Causes the standard I/O file descriptors (0, 1, and 2) take their values from the current I/O ports. It is exactly equivalent to the series of redirections:

```
(dup (current-input-port)  0)
(dup (current-output-port) 1)
(dup (error-output-port)   2)
```

```
(with-stdio-ports* thunk) → any
  thunk : (-> any)
(with-stdio-ports body ...+)
```

Why not `move->fdes`? Because the current output port and error port might be the same port.

with-stdio-ports* binds the standard ports (current-input-port), (current-output-port), and (error-output-port) to be ports on file descriptors 0, 1, 2, and then calls *thunk*. It is equivalent to:

```
(with-current-input-port (fdes->inport 0)
  (with-current-output-port (fdes->inport 1)
    (with-error-output-port (fdes->outport 2)
      (thunk))))
```

The with-stdio-ports special form is merely syntactic sugar.

3.2.3   String ports

Scheme48 has string ports, which you can use. Scsh has not committed to the particular interface or names that scheme48 uses, so be warned that the interface described herein may be liable to change.

```
(make-string-input-port string) → port?
  string : string?
```

Returns a port that reads characters from the supplied string.

```
(make-string-output-port) → port?
(string-output-port-output port) → string?
  port : port?
```

20

A string output port is a port that collects the characters given to it into a string. The accumulated string is retrieved by applying `string-output-port-output` to the port.

```
(call-with-string-output-port procedure) → string?
  procedure : (-> port? any)
```

The *procedure* value is called on a port. When it returns, `call-with-string-output-port` returns a string containing the characters that were written to that port during the execution of *procedure*.

3.2.4   Revealed Ports and File Descriptors

The material in this section and the following one is not critical for most applications. You may safely skim or completely skip this section on a first reading.

Dealing with Unix file descriptors in a Scheme environment is difficult. In Unix, open files are part of the process environment, and are referenced by small integers called *file descriptors*. Open file descriptors are the fundamental way I/O redirections are passed to subprocesses, since file descriptors are preserved across fork's and exec's.

Scheme, on the other hand, uses ports for specifying I/O sources. Ports are garbage-collected Scheme objects, not integers. Ports can be garbage collected; when a port is collected, it is also closed. Because file descriptors are just integers, it's impossible to garbage collect them—you wouldn't be able to close file descriptor 3 unless there were no 3's in the system, and you could further prove that your program would never again compute a 3. This is difficult at best.

If a Scheme program only used Scheme ports, and never actually used file descriptors, this would not be a problem. But Scheme code must descend to the file descriptor level in at least two circumstances:

- when interfacing to foreign code
- when interfacing to a subprocess

This causes a problem. Suppose we have a Scheme port constructed on top of file descriptor 2. We intend to fork off a program that will inherit this file descriptor. If we drop references to the port, the garbage collector may prematurely close file 2 before we fork the subprocess. The interface described below is intended to fix this and other problems arising from the mismatch between ports and file descriptors.

The scsh kernel maintains a port table that maps a file descriptor to the scheme port allocated for it (or, #f if there is no port allocated for this file descriptor). This is used to ensure that there is at most one open port for each open file descriptor.

Conceptually, the port data structure for file ports has two fields besides the descriptor: `revealed` and `closed?`. When a file port is closed with `(close port)`, the port's file descriptor is closed, its entry in the port table is cleared, and the port's `closed?` field is set to true.

When a file descriptor is closed with `(close fdes)`, any associated port is shifted to a new file descriptor created with `(dup fdes)`. The port has its revealed count reset to zero (and hence becomes eligible for closing on GC). See discussion below. To really put a stake through a descriptor's heart without waiting for associated ports to be GC'd, you must say one of

```
(close (fdes->inport fdes))
(close (fdes->output fdes))
```

The `revealed` field is an aid to garbage collection. It is an integer semaphore. If it is zero, the port's file descriptor can be closed when the port is collected. Essentially, the `revealed` field reflects whether or not the port's file descriptor has escaped to the scheme user. If the scheme user doesn't know what file descriptor is associated with a given port, then he can't possibly retain an "integer handle" on the port after dropping pointers to the port itself, so the garbage collector is free to close the file.

Ports allocated with `open-output-file` and `open-input-file` are unrevealed ports—i.e., `revealed` is initialised to 0. No one knows the port's file descriptor, so the file descriptor can be closed when the port is collected.

The functions `fdes->output-port`, `fdes->input-port`, `port->fdes` are used to shift back and forth between file descriptors and ports. When `port->fdes` reveals a port's file descriptor, it increments the port's `revealed` field. When the user is through with the file descriptor, he can call `(release-port-handle port)`, which decrements the count. The function `(call/fdes fd/port proc)` automates this protocol. `call/fdes` uses `dynamic-wind` to enforce the protocol. If *proc* throws out of the `call/fdes` application, the unwind handler releases the descriptor handle; if the user subsequently tries to throw *back* into *proc*'s context, the wind handler raises an error. When the user maps a file descriptor to a port with `fdes->outport` or `fdes->inport`, the port has its revealed field incremented.

Not all file descriptors are created by requests to make ports. Some are inherited on process invocation via exec(2)}, and are simply part of the global environment. Subprocesses may depend upon them, so if a port is later allocated for these file descriptors, is should be considered as a revealed port. For example, when the scheme shell's process starts up, it opens ports on file descriptors 0, 1, and 2 for the initial values of `(current-input-port)`, `(current-output-port)`, and `(error-output-port)`. These ports are initialised with `revealed` set to 1, so that stdin, stdout, and stderr are not closed even if the user drops the port.

Unrevealed file ports have the nice property that they can be closed when all pointers to the port are dropped. This can happen during gc, or at an `exec()`—since all memory is

dropped at an `exec()`. No one knows the file descriptor associated with the port, so the exec'd process certainly can't refer to it.

This facility preserves the transparent close-on-collect property for file ports that are used in straightforward ways, yet allows access to the underlying Unix substrate without interference from the garbage collector. This is critical, since shell programming absolutely requires access to the Unix file descriptors, as their numerical values are a critical part of the process interface.

A port's underlying file descriptor can be shifted around with `dup(2)` when convenient. That is, the actual file descriptor on top of which a port is constructed can be shifted around underneath the port by the scsh kernel when necessary. This is important, because when the user is setting up file descriptors prior to a `exec(2)`, he may explicitly use a file descriptor that has already been allocated to some port. In this case, the scsh kernel just shifts the port's file descriptor to some new location with `dup`, freeing up its old descriptor. This prevents errors from happening in the following scenario. Suppose we have a file open on port `f`. Now we want to run a program that reads input on file 0, writes output to file 1, errors to file 2, and logs execution information on file 3. We want to run this program with input from `f`. So we write:

```
(run (/usr/shivers/bin/prog)
     (> 1 output.txt)
     (> 2 error.log)
     (> 3 trace.log)
     (= 0 ,f))
```

Now, suppose by ill chance that, unbeknownst to us, when the operating system opened `f`'s file, it allocated descriptor 3 for it. If we blindly redirect `trace.log` into file descriptor 3, we'll clobber `f`! However, the port-shuffling machinery saves us: when the `run` form tries to dup `trace.log`'s file descriptor to 3, `dup` will notice that file descriptor 3 is already associated with an unrevealed port (i.e., `f`). So, it will first move `f` to some other file descriptor. This keeps `f` alive and well so that it can subsequently be dup'd into descriptor 0 for `prog`'s stdin.

The port-shifting machinery makes the following guarantee: a port is only moved when the underlying file descriptor is closed, either by a `close()` or a `dup2()` operation. Otherwise a port/file-descriptor association is stable.

Under normal circumstances, all this machinery just works behind the scenes to keep things straightened out. The only time the user has to think about it is when he starts accessing file descriptors from ports, which he should almost never have to do. If a user starts asking what file descriptors have been allocated to what ports, he has to take responsibility for managing this information.

3.2.5 Port-Mapping Machinery

The procedures provided in this section are almost never needed. You may safely skim or completely skip this section on a first reading.

Here are the routines for manipulating ports in scsh. The important points to remember are:

- A file port is associated with an open file, not a particular file descriptor.

- The association between a file port and a particular file descriptor is never changed *except* when the file descriptor is explicitly closed. "Closing" includes being used as the target of a `dup2`, so the set of procedures below that close their targets are `close`, two-argument `dup`, and `move->fdes`. If the target file descriptor of one of these routines has an allocated port, the port will be shifted to another freshly-allocated file descriptor, and marked as unrevealed, thus preserving the port but freeing its old file descriptor.

These rules are what is necessary to "make things work out" with no surprises in the general case.

```
(fdes->inport fd) → port?
  fd : integer?
(fdes->outport fd) → port?
  fd : integer?
```

These guys return an input or output port respecitvely, backed by `fd`. The returned port has its revealed count set to 1.

```
(port->fdes port) → integer?
  port : fdport?
```

Returns the file descriptor backing `port` and increments its revealed count by 1.

```
(port-revealed port) → (or/c integer? #f)
  port : fdport?
```

Return the port's revealed count if positive, otherwise #f.

```
(release-port-handle port) → undefined
  port : fdport?
```

Decrement `port`'s revealed count.

```
(call/fdes fd/port consumer) → any
  fd/port : (or/c integer? fdport?)
  consumer : (-> integer? any)
```

Calls *consumer* on a file descriptor; takes care of revealed bookkeeping. If *fd/port* is a file descriptor, this is just `(consumer fd/port)`. If *fd/port* is a port, calls *consumer* on its underlying file descriptor. While *consumer* is running, the port's revealed count is incremented.

When `call/fdes` is called with port argument, you are not allowed to throw into *consumer* with a stored continuation, as that would violate the revealed-count bookkeeping.

```
(move->fdes fd/port target-fd) → (or/c integer? fdport?)
  fd/port : (or/c integer? fdport?)
  target-fd : integer?
```

Maps fd -> fd and port ->port.

If *fd/port* is a file-descriptor not equal to `target-fd`, dup it to `target-fd` and close it. Returns `target-fd`.

If *fd/port* is a port, it is shifted to `target-fd`, by duping its underlying file-descriptor if necessary. *fd/port*'s original file descriptor is closed (if it was different from `target-fd`). Returns the port. This operation resets *fd/port*'s revealed count to 1.

In all cases when *fd/port* is actually shifted, if there is a port already using `target-fd`, it is first relocated to some other file descriptor.

3.2.6 Unix I/O

```
(dup fd/port) → fdport?
  fd/port : (or/c integer? fdport?)
(dup fd/port newfd) → fdport?
  fd/port : (or/c integer? fdport?)
  newfd : integer?
(dup->inport fd/port) → fdport?
  fd/port : (or/c integer? fdport?)
(dup->inport fd/port newfd) → fdport?
  fd/port : (or/c integer? fdport?)
  newfd : integer?
(dup->outport fd/port) → fdport?
  fd/port : (or/c integer? fdport?)
(dup->outport fd/port newfd) → fdport?
  fd/port : (or/c integer? fdport?)
  newfd : integer?
(dup->fdes fd/port) → integer?
  fd/port : (or/c integer? fdport?)
(dup->fdes fd/port newfd) → integer?
  fd/port : (or/c integer? fdport?)
  newfd : integer?
```

These procedures provide the functionality of C's `dup()` and `dup2()`. The different routines return different types of values: `dup->inport`, `dup->outport`, and `dup->fdes` return input ports, output ports, and integer file descriptors, respectively. `dup`'s return value depends on on the type of `fd/port`—it maps fd -> fd and port -> port.

These procedures use the Unix `dup()` syscall to replicate the file descriptor or file port `fd/port`. If a `newfd` file descriptor is given, it is used as the target of the dup operation, i.e., the operation is a `dup2()`. In this case, procedures that return a port (such as `dup->inport`) will return one with the revealed count set to one. For example, `(dup (current-input-port) 5)` produces a new port with underlying file descriptor 5, whose revealed count is 1. If `newfd` is not specified, then the operating system chooses the file descriptor, and any returned port is marked as unrevealed.

If the `newfd` target is given, and some port is already using that file descriptor, the port is first quietly shifted (with another `dup`) to some other file descriptor (zeroing its revealed count).

Since scheme doesn't provide read/write ports, `dup->inport` and `dup->outport` can be useful for getting an output version of an input port, or *vice versa*. For example, if `p` is an input port open on a tty, and we would like to do output to that tty, we can simply use `(dup->outport p)` to produce an equivalent output port for the tty. Be sure to open the file with the `open/read+write` flag for this.

```
(seek fd offset) → integer?
  fd : integer?
  offset : integer?
(seek fd offset whence) → integer?
  fd : integer?
  offset : integer?
  whence : integer?
```

Reposition the I/O cursor for a file descriptor. `whence` is one of `seek/set`, `seek/delta`, or `seek/end`, and defaults to `seek/set`. If `seek/set`, then `offset` is an absolute index into the file; if `seek/delta`, then `offset` is a relative offset from the current I/O cursor; if `seek/end`, then `offset` is a relative offset from the end of file. Note that not all file descriptors are seekable; this is dependent on the OS implementation. The return value is the resulting position of the I/O cursor in the I/O stream.

```
(tell fd) → integer?
  fd : integer?
```

Returns the position of the I/O cursor in the the I/O stream. Not all file descriptors support cursor-reporting; this is dependent on the OS implementation.

```
(open-file fname options [mode]) → fdport?
  fname : string?
  options : file-options?
  mode : file-mode? = (filemode read write)
```

26

options is a file-option? (ref). You must use exactly one of the options `read-only`, `write-only`, or `read-write`. The returned port is an input port if the *options* permit it, otherwise an output port. scheme48/scsh do not have input/output ports, so it's one or the other. You can hack simultaneous I/O on a file by opening it r/w, taking the result input port, and duping it to an output port with `dup->outport`.

```
(open-input-file fname [options]) → port?
  fname : string?
  options : file-options? = (file-options)
(open-output-file fname [options mode]) → port?
  fname : string?
  options : file-options? = (file-options create truncate)
  mode : file-mode? = (file-mode read write)
```

These are equivalent to `open-file`, after first including `read-only` or `write-only` options, respectively, in the *options* argument. The default values for *options* make the procedures backwards-compatible with their unary RnRS definitions.

```
(pipe) → [rport fdport?] [wport fdport?]
```

Returns two ports *rport* and *wport*, the read and write endpoints respectively of a Unix pipe.

3.2.7 Buffered I/O

```
(force-output fd/port) → undefined
  fd/port : (or/c integer? fdport?)
```

This procedure does nothing when applied to an integer file descriptor. It flushes buffered output when applied to a port, and raises a write-error exception on error. Returns no value.

```
(flush-all-ports) → undefined
```

This procedure flushes all open output ports with buffered data.

3.3 Filesystem

Besides the procedures in this section, which allow access to the computer's file system, scsh also provides a set of procedures which manipulate file *names*. These string-processing procedures are documented in

3.3.1   Manipulating Filesystem Objects

```
(create-directory fname [mode override?]) → undefined
  fname : string?
  mode : file-mode? = (file-mode all)
  override? : (or/c #f 'query any/c) = #f
(create-fifo fname [mode override?]) → undefined
  fname : string?
  mode : file-mode? = (file-mode all)
  override? : (or/c #f 'query any/c) = #f
(create-hard-link oldname newname [override?]) → undefined
  oldname : string?
  newname : string?
  override? : (or/c #f 'query any/c) = #f
(create-symlink oldname newname [override?]) → undefined
  oldname : string?
  newname : string?
  override? : (or/c #f 'query any/c) = #f
```

These procedures create objects of various kinds in the file system. The *override?* argument controls the action if there is already an object in the file system with the new name:

#f	signal an error (default)
'query	prompt the user
other	delete the old object (with `delete-file` or `delete-directory` as appropriate) before creating the new object

mode defaults to `(file-mode all)` (but is masked by the current umask).

```
(delete-directory fname) → undefined
  fname : string?
(delete-file fname) → undefined
  fname : string?
(delete-filesys-object fname) → undefined
  fname : string?
```

These procedures delete objects from the filesystem. The `delete-filesys-object` procedure will delete an object of any type from the file system: files, (empty) directories, symlinks, fifos, etc.

If the object being deleted doesn't exist, `delete-directory` and `delete-file` raise an error, while delete-filesys-object simply returns.

```
(read-symlink fname) → string?
  fname : string?
```

Return the filename referenced by the symbolic link *fname*.

```
(rename-file old-fname new-fname [override?]) → undefined
  old-fname : string?
  new-fname : string?
  override? : (or/c #f 'query any/c) = #f
```

If you override an existing object, then *old-fname* and *new-fname* must type-match— either both directories, or both non-directories. This is required by the semantics of Unix `rename()`.

```
(set-file-mode fname/fd/port mode) → undefined
  fname/fd/port : (or/c string? integer? fdport?)
  mode : file-mode?
(set-file-owner fname/fd/port uid) → undefined
  fname/fd/port : (or/c string? integer? fdport?)
  uid : integer?
(set-file-group fname/fd/port gid) → undefined
  fname/fd/port : (or/c string? integer? fdport?)
  gid : integer?
```

These procedures set the permission bits, owner id, and group id of a file, respectively. The file can be specified by giving the file name, or either an integer file descriptor or a port open on the file. Setting file user ownership usually requires root privileges.

```
(set-file-times fname [access-time mod-time]) → undefined
  fname : string?
  access-time : integer? = (current-time)
  mod-time : integer? = (current-time)
```

fname to the supplied values (see (link to sec:time) for the scsh representation of time). If neither time argument is supplied, they are both taken to be the current time. You must provide both times or neither. If the procedure completes successfully, the file's time of last status-change (`ctime`) is set to the current time.

```
(sync-file fd/port) → undefined
  fd/port : (or/c integer? fdport?)
(sync-file-system) → undefined
```

Calling `sync-file` causes Unix to update the disk data structures for a given file. If *fd/port* is a port, any buffered data it may have is first flushed. Calling `sync-file-system` synchronises the kernel's entire file system with the disk.

```
(truncate-file fname/fd/port len) → undefined
  fname/fd/port : (or/c string? integer? fdport?)
  len : integer?
```

Truncate the specified file to *len* bytes in length.

3.3.2 Querying File Information

```
(file-info fname/fd/port [chase?]) → file-info?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
```

Returns a record structure containing everything there is to know about a file. If the *chase?* flag is true (the default), then the procedure chases symlinks and reports on the files to which they refer. If *chase?* is false, then the procedure checks the actual file itself, even if it's a symlink. The *chase?* flag is ignored if the file argument is a file descriptor or port.

The value returned is a *file-info record*, whose accessors are defined in this chapter.

```
(file-info:type finfo)
 → (or/c 'block-special 'char-special 'directory 'fifo 'regular 'socket 'symlink)
  finfo : file-info?
```

Returns a symbol denoting the type of the file described by *finfo*.

```
(file-info:device finfo) → integer?
  finfo : file-info?
```

Returns an integer denoting the device that the file described by *finfo* resides on.

```
(file-info:inode finfo) → integer?
  finfo : file-info?
```

Returns an integer denoting the file system inode that points to the file described by *finfo*.

```
(file-info:mode finfo) → file-mode?
  finfo : file-info?
```

Returns a file-mode object describing the permissions set on the file described by *finfo*.

```
(file-info:nlinks finfo) → integer?
  finfo : file-info?
```

Returns the number of hard links to the file described by *finfo*.

```
(file-info:uid finfo) → integer?
  finfo : file-info?
```

Returns user id of the owner of the file file described by *finfo*.

```
(file-info:gid finfo) → integer?
  finfo : file-info?
```

Returns the group id of the file described by *finfo*.

```
(file-info:size finfo) → integer?
  finfo : file-info?
```

Returns the size in bytes of the file described by *finfo*.

```
(file-info:size finfo) → integer?
  finfo : file-info?
```

Returns the time at which the file described by *finfo* was last accessed.

```
(file-info:atime finfo) → integer?
  finfo : file-info?
```

Returns the time at which the file described by *finfo* was last accessed.

```
(file-info:mtime finfo) → integer?
  finfo : file-info?
```

Returns the time at which the file described by *finfo* was last modified.

```
(file-info:ctime finfo) → integer?
  finfo : file-info?
```

Returns the time at which the file described by *finfo* last had its status changed.

```
(file:type fname/fd/port [chase?])
 → (or/c 'block-special 'char-special 'directory 'fifo 'regular 'socket 'symlink)
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:device fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:inode fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:mode fname/fd/port [chase?]) → file-mode?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
```

```
(file:nlinks fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:uid fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:gid fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:size fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:atime fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:mtime fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file:ctime fname/fd/port [chase?]) → integer?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
```

These procuders are a composition of `file-info` and its accessors. They allow more convenient access to file based information, without handling an intermediary file-info object.

```
(file-directory? fname/fd/port [chase?]) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file-fifo? fname/fd/port [chase?]) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file-regular? fname/fd/port [chase?]) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file-socket? fname/fd/port [chase?]) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file-special? fname/fd/port [chase?]) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
  chase? : any/c = #t
(file-symlink? fname/fd/port) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
```

These procedures are file-type predicates that test the type of a given file. They are applied to the same arguments to which `file-info` is applied; the sole exception is `file-symlink?`, which does not take the optional *chase?* second argument.

```
(file-info-directory? file-info) → boolean?
  file-info : file-info?
(file-info-fifo? file-info) → boolean?
  file-info : file-info?
(file-info-regular? file-info) → boolean?
  file-info : file-info?
(file-info-socket? file-info) → boolean?
  file-info : file-info?
(file-info-special? file-info) → boolean?
  file-info : file-info?
(file-info-symlink? file-info) → boolean?
  file-info : file-info?
```

These are variants of the file-type predicates which work directly on `file-info` records.

```
(file-not-readable? fname/fd/port)
 → (or/c #f 'search-denied 'permission 'no-directory 'nonexistent)
  fname/fd/port : (or/c string? integer? fdport?)
(file-not-writable? fname/fd/port)
 → (or/c #f 'search-denied 'permission 'no-directory 'nonexistent)
  fname/fd/port : (or/c string? integer? fdport?)
(file-not-executable? fname/fd/port)
 → (or/c #f 'search-denied 'permission 'no-directory 'nonexistent)
  fname/fd/port : (or/c string? integer? fdport?)
```

This set of procedures are a convenient means to work on the permission bits of a file. The meaning of their return values are as follows:

#f	Access permitted
'search-denied	Can't stat — a protected directory is blocking access
'permission	Permission denied.
'no-directory	Some directory doesn't exist.
'nonexistent	File doesn't exist.

A file is considered writeable if either (1) it exists and is writeable or (2) it doesn't exist and the directory is writeable. Since symlink permission bits are ignored by the filesystem, these calls do not take a *chase?* flag.

Note that these procedures use the process' *effective* user and group ids for permission checking. Posix defines an `access()` function that uses the process' real uid and gids. This is handy for setuid programs that would like to find out if the actual user has specific rights; scsh ought to provide this functionality (but doesn't at the current time).

There are several problems with these procedures. First, there's an atomicity issue. In between checking permissions for a file and then trying an operation on the file, another

process could change the permissions, so a return value from these functions guarantees nothing. Second, the code special-cases permission checking when the uid is root—if the file exists, root is assumed to have the requested permission. However, not even root can write a file that is on a read-only file system, such as a CD ROM. In this case, `file-not-writable?` will lie, saying that root has write access, when in fact the opening the file for write access will fail. Finally, write permission confounds write access and create access. These should be disentangled.

Some of these problems could be avoided if Posix had a real-uid variant of the `access()` call we could use, but the atomicity issue is still a problem. In the final analysis, the only way to find out if you have the right to perform an operation on a file is to try and open it for the desired operation. These permission-checking functions are mostly intended for script-writing, where loose guarantees are tolerated.

```
(file-readable? fname/fd/port) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
(file-writable? fname/fd/port) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
(file-executable? fname/fd/port) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
```

These procedures are the logical negation of the preceding `file-not-*` procedures. Refer to those for a discussion of their problems and limitations. These procedures will only ever return #t or #f, and not the symbols giving specific reasons.

```
(file-info-not-readable? file-info) → boolean?
  file-info : file-info?
(file-info-not-writable? file-info) → boolean?
  file-info : file-info?
(file-info-not-executable? file-info) → boolean?
  file-info : file-info?
(file-info-readable? file-info) → boolean?
  file-info : file-info?
(file-info-writable? file-info) → boolean?
  file-info : file-info?
(file-info-executable? file-info) → boolean?
  file-info : file-info?
```

There are variants of the file permission predicates which work directly on `file-info` records.

```
(file-not-exists? fname/fd/port)
 → (or/c boolean? 'search-denied)
  fname/fd/port : (or/c string? integer? fdport?)
(file-exists? fname/fd/port) → boolean?
  fname/fd/port : (or/c string? integer? fdport?)
```

The meaning of the return values of `file-not-exists?` are as follows:

`#f`	Exists
`#t`	Doesn't exist.
`'search-denied`	Some protected directory is blocking the search.

`file-exists?` is simply the logical negation of `file-not-exists?`.

3.3.3   Traversing Directories

```
(directory-files [dir dotfiles?]) → (listof string?)
  dir : string? = (cwd)
  dotfiles? : boolean? = #f
```

Return the list of files in directory `dir`, which defaults to the current working directory. The `dotfiles?` flag causes dot files to be included in the list. Regardless of the value of `dotfiles?`, the two files `"."` and `".."` are *never* returned.

The directory `dir` is not prepended to each file name in the result list. That is,

```
(directory-files "/etc")
```

returns

```
("chown" "exports" "fstab" ...)
```

not

```
("/etc/chown" "/etc/exports" "/etc/fstab" ...)
```

To use the files in returned list, the programmer can either manually prepend the directory:

```
(map (lambda (f) (string-append dir "/" f)) files)
```

or cd to the directory before using the file names:

```
(with-cwd dir
  (for-each delete-file (directory-files)))
```

or use the `glob` procedure, defined in this chapter.

A directory list can be generated by `(run/strings (ls))`, but this is unreliable, as filenames with whitespace in their names will be split into separate entries. `directory-files` is reliable.

```
(open-directory-stream dir) → directory-stream?
  dir : string?
(directory-stream? maybe-directory-stream) → boolean?
  maybe-directory-stream : any/c
(read-directory-stream directory-stream) → (or/c string? #f)
  directory-stream : directory-stream?
(close-directory-stream directory-stream) → undefined
  directory-stream : directory-stream?
```

These functions implement a direct interface to the `opendir()` / `readdir()` / `closedir()` family of functions for processing directory streams. `(open-directory-stream dir)` creates a stream of files in the directory `dir`. `directory-stream?` is a predicate that recognizes directory-streams. `(read-directory-stream directory-stream)` returns the next file in the stream or `#f` if no such file exists. Finally, `(close-directory-stream directory-stream)` closes the stream.

3.3.4 Globbing

```
(glob pattern ...) → (listof string)
  pattern : string?
```

Glob each pattern against the filesystem and return the sorted list. Duplicates are not removed. Patterns matching nothing are not included literally.

C shell {a,b,c} patterns are expanded. Backslash quotes characters, turning off the special meaning of {, }, *, [,], and ?.

Note that the rules of backslash for Scheme strings and glob patterns work together to require four backslashes in a row to specify a single literal backslash. Fortunately, it is very rare that a backslash occurs in a Unix file name.

A glob subpattern will not match against dot files unless the first character of the subpattern is a literal ".". Further, a dot subpattern will not match the files "." or ".." unless it is a constant pattern, as in `(glob "../*/*.c")`. So a directory's dot files can be reliably generated with the simple glob pattern ".*".

Some examples.

All the C and #include files in my directory:

```
(glob "*.c" "*.h")
```

All the C files in this directory and its immediate subdirectories:

```
(glob "*.c" "*/*.c")
```

All the C files in the lexer and parser dirs:

```
(glob "lexer/*.c" "parser/*.c")
(glob "{lexer,parser}/*.c")
```

All the C files in the strange directory `"{lexer,parser}"`:

```
(glob "\\{lexer,parser\\}/*.c")
```

All the files ending in `"*"`, e.g. `("foo*" "bar*")`:

```
(glob "*\\*")
```

All files containing the string `"lexer"`, e.g. `("mylexer.c" "lexer1.notes")`:

```
(glob "*lexer*")
```

Either `("lexer")` or `()`:

```
(glob "lexer")
```

If the first character of the pattern (after expanding braces) is a slash, the search begins at root; otherwise, the search begins in the current working directory.

If the last character of the pattern (after expanding braces) is a slash, then the result matches must be directories, e.g.

```
(glob "/usr/man/man?/") => ("/usr/man/man1/" "/usr/man/man2/" ...)
```

Globbing can sometimes be useful when we need a list of a directory's files where each element in the list includes the pathname for the file.

Compare:

```
(directory-files "../include") => ("cig.h" "decls.h" ...)

(glob "../include/*") => ("../include/cig.h" "../include/decls.h" ...)
```

(glob-quote *pattern*) → string?
  pattern : string?

Returns a constant glob pattern that exactly matches *pattern*. All wild-card characters in *pattern* are quoted with a backslash.

```
(glob-quote "Any *.c files?") => "Any \\*.c files\\?"
```

3.3.5 Temporary Files

```
(create-temp-file [prefix]) → string?
  prefix : string? = (fluid *temp-file-template*)
```

`create-temp-file` creates a new temporary file and return its name. The optional argument specifies the filename prefix to use, and defaults to the value of `"$TMPDIR/<pid>"` if `$TMPDIR` is set and to `"/var/tmp/<pid>"` otherwise, where *pid* is the current process' id. The procedure generates a sequence of filenames that have *prefix* as a common prefix, looking for a filename that doesn't already exist in the file system. When it finds one, it creates it, with permission `(file-mode owner-read owner-write)` and returns the filename. (The file permission can be changed to a more permissive permission with `set-file-mode` after being created).

This file is guaranteed to be brand new. No other process will have it open. This procedure does not simply return a filename that is very likely to be unused. It returns a filename that definitely did not exist at the moment `create-temp-file` created it.

It is not necessary for the process' pid to be a part of the filename for the uniqueness guarantees to hold. The pid component of the default prefix simply serves to scatter the name searches into sparse regions, so that collisions are less likely to occur. This speeds things up, but does not affect correctness.

Security note: doing I/O to files created this way in `"/var/tmp/"` is not necessarily secure. General users have write access to `"/var/tmp/"`, so even if an attacker cannot access the new temp file, he can delete it and replace it with one of his own. A subsequent open of this filename will then give you his file, to which he has access rights. There are several ways to defeat this attack,

- Use `temp-file-iterate`, to return the file descriptor allocated when the file is opened. This will work if the file only needs to be opened once.

- If the file needs to be opened twice or more, create it in a protected directory, e.g., `"$HOME"`

- Ensure that `"/var/tmp"` has its sticky bit set. This requires system administrator privileges

The actual default prefix used is controlled by the dynamic variable `*temp-file-template*`, and can be overridden for increased security. See `temp-file-iterate` for details..

```
(temp-file-iterate maker [template]) → any/c ...
  maker : (-> string? (values any/c ...))
  template : string? = (fluid *temp-file-template*)
*temp-file-template* : string?
```

`temp-file-iterate` can be used to perform certain atomic transactions on the file system involving filenames. Some examples:

- Linking a file to a fresh backup temp name.

- Creating and opening an unused, secure temp file.

- Creating an unused temporary directory.

This procedure uses *template* to generate a series of trial file names. *template* should be a `format` control string and its default is taken from the value of the dynamic variable `*temp-file-template*` which itself defaults to `"$TMPDIR/<pid>.~a"` if `$TMPDIR` is set and `"/usr/tmp/<pid>.~a"` otherwise, where *pid* is the scsh process' pid. File names are generated by calling `format` to instantiate the template's `"~a"` field with a varying string. For increased security, a user may wish to change the template to use a directory not allowing world write access (e.g., his home directory). Scsh uses scheme48's `fluids` package to implement dynamic binding; see the documentation for that for details.

maker is a procedure which is serially called on each file name generated. It must return at least one value; it may return multiple values. If the first return value is `#f` or if *maker* raises the `errno/exist` errno exception, `temp-file-iterate` will loop, generating a new file name and calling *maker* again. If the first return value is true, the loop is terminated, returning whatever value(s) *maker* returned.

After a number of unsuccessful trials, `temp-file-iterate` may give up and signal an error.

Thus, if we ignore its optional *prefix* argument, `create-temp-file` could be defined as:

```
(define (create-temp-file)
  (let ((options (file-options create exclusive))
        (mode (file-mode owner-read owner-write)))
    (temp-file-iterate
      (lambda (f)
        (close (open-output-file f options mode)) f))))
```

To rename a file to a temporary name:

```
(temp-file-iterate
  (lambda (backup)
    (create-hard-link old-file backup) backup)
      ".#temp.~a") ; Keep link in cwd.
(delete-file old-file)
```

Recall that scsh reports syscall failure by raising an error exception, not by returning an error code. This is critical to to this example—the programmer can assume that if the `temp-file-iterate` call returns, it returns successfully. So the following `delete-file` call can be reliably invoked, safe in the knowledge that the backup link has definitely been established.

To create a unique temporary directory:

```
(temp-file-iterate
  (lambda (dir) (create-directory dir) dir)
  "/var/tmp/tempdir.~a")
```

Similar operations can be used to generate unique symlinks and fifos, or to return values other than the new filename (e.g., an open file descriptor or port).

```
(temp-file-channel) → [input-port port?] [output-port port?]
```

This procedure can be used to provide an interprocess communications channel with arbitrary-sized buffering. It returns two values, an input port and an output port, both open on a new temp file. The temp file itself is deleted from the Unix file tree before `temp-file-channel` returns, so the file is essentially unnamed, and its disk storage is reclaimed as soon as the two ports are closed.

`temp-file-channel` is analogous to `port-pipe` with two exceptions:

- If the writer process gets ahead of the reader process, it will not hang waiting for some small pipe buffer to drain. It will simply buffer the data on disk. This is good.

- If the reader process gets ahead of the writer process, it will also not hang waiting for data from the writer process. It will simply see and report an end of file. This is bad.

In order to ensure that an end-of-file returned to the reader is legitimate, the reader and writer must serialise their I/O. The simplest way to do this is for the reader to delay doing input until the writer has completely finished doing output, or exited.

3.4 Processes

```
(exec program arg ...) → any
  program : string?
  arg : string?
(exec-path program arg ...) → any
  program : string?
  arg : string?
(exec/env program environment arg ...) → any
  program : string?
  environment : (or/c (listof (cons/c string? string?)) #t)
  arg : string?
(exec-path/env program environment arg ...) → any
  program : string?
  environment : (or/c (listof (cons/c string? string?)) #t)
  arg : string?
```

The `.../env` variants take an environment specified as a string to string alist. An environment of `#t` is taken to mean the current process' environment (i.e., the value of the external char `**environ`).

[Rationale: `#f` is a more convenient marker for the current environment than `#t`, but would cause an ambiguity on Schemes that identify `#f` and `()`.]

The path-searching variants search the directories in the list `exec-path-list` for the program. A path-search is not performed if the program name contains a slash character—it is used directly. So a program with a name like `"bin/prog"` always executes the program `bin/prog` in the current working directory. See `$path` and `exec-path-list`, below.

Note that there is no analog to the C function `execv()`. To get the effect just do

```
(apply exec prog arglist)
```

All of these procedures flush buffered output and close unrevealed ports before executing the new binary. To avoid flushing buffered output, see `%exec` below.

Note that the C `exec()` procedure allows the zeroth element of the argument vector to be different from the file being executed, e.g.

```
char *argv[] = {"-", "-f", 0};
exec("/bin/csh", argv, envp);
```

The scsh `exec`, `exec-path`, `exec/env`, and `exec-path/env` procedures do not give this functionality—element 0 of the arg vector is always identical to the `prog` argument. In the rare case the user wishes to differentiate these two items, he can use the low-level `%exec` and `exec-path-search` procedures.

These procedures never return under any circumstances. As with any other system call, if there is an error, they raise an exception.

```
(%exec program arglist env) → any
  program : string?
  arglist : (listof string?)
  env : (or/c #t (listof (pair/c string? string?)))
(exec-path-search fname pathlist) → (or/c string? #f)
  fname : string?
  pathlist : (listof string?)
```

The `%exec` procedure is the low-level interface to the system call. The `arglist` parameter is a list of arguments; `env` is either a string to string alist or `#t`. The new program's `argv[0]` will be taken from `(car arglist)`, *not* from `prog`. An environment of `#t` means the current process' environment. `%exec` does not flush buffered output (see `flush-all-ports`).

All exec procedures, including `%exec`, coerce the *prog* and *arg* values to strings using the usual conversion rules: numbers are converted to decimal numerals, and symbols converted to their print-names.

`exec-path-search` searches the directories of *pathlist* looking for an occurrence of file `fname`. If no executable file is found, it returns `#f`. If *fname* contains a slash character, the path search isshort-circuited, but the procedure still checks to ensure that the file exists and is executable—if not, it still returns `#f`. Users of this procedure should be aware that it invites a potential race condition: between checking the file with `exec-path-search` and executing it with `%exec`, the file's status might change. The only atomic way to do the search is to loop over the candidate file names, exec'ing each one and looping when the exec operation fails.

See `$path` and `exec-path-list`, below.

```
(exit [status]) → any
  status : integer? = 0
(%exit [status]) → any
  status : integer? = 0
```

These procedures terminate the current process with a given exit status. The default exit status is 0. The low-level `%exit` procedure immediately terminates the process without flushing buffered output.

```
(call-terminally thunk) → any
  thunk : (-> (values any ...))
```

`call-terminally` calls its thunk. When the thunk returns, the process exits. Although `call-terminally` could be implemented as

```
(lambda (thunk) (thunk) (exit 0))
```

an implementation can take advantage of the fact that this procedure never returns. For example, the runtime can start with a fresh stack and also start with a fresh dynamic environment, where shadowed bindings are discarded. This can allow the old stack and dynamic environment to be collected (assuming this data is not reachable through some live continuation).

```
(suspend) → undefined
```

Suspend the current process with a SIGSTOP signal.

```
(fork thunk [continue-threads?]) → (or/c proc? #f)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
(%fork thunk [continue-threads?]) → (or/c proc? #f)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
```

`fork` with no arguments or `#f` instead of a thunk is like C `fork()`. In the parent process, it returns the child's *process object* (see below for more information on process objects). In the child process, it returns `#f`.

`fork` with an argument only returns in the parent process, returning the child's process object. The child process calls *thunk* and then exits.

`fork` flushes buffered output before forking, and sets the child process to non-interactive. `%fork` does not perform this bookkeeping; it simply forks.

The optional boolean argument *continue-threads?* specifies whether the currently active threads continue to run in the child or not. The default is `#f`.

```
(fork/pipe thunk [continue-threads?]) → (or/c proc? #f)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
(%fork/pipe thunk [continue-threads?]) → (or/c proc? #f)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
```

Like `fork` and `%fork`, but the parent and child communicate via a pipe connecting the parent's stdin to the child's stdout. These procedures side-effect the parent by changing his stdin.

In effect, `fork/pipe` splices a process into the data stream immediately upstream of the current process. This is the basic function for creating pipelines. Long pipelines are built by performing a sequence of `fork/pipe` calls. For example, to create a background two-process pipe `"a | b"`, we write:

```
(fork (lambda () (fork/pipe a) (b)))
```

which returns the process object for `b`'s process.

To create a background three-process pipe `"a | b | c"`, we write:

```
(fork (lambda () (fork/pipe a)
                 (fork/pipe b)
                 (c)))
```

which returns the process object for `c`'s process.

Note that these procedures affect file descriptors, not ports. That is, the pipe is allocated connecting the child's file descriptor 1 to the parent's file descriptor 0. *Any previous Scheme port built over these affected file descriptors is shifted to a new, unused file descriptor with* `dup` *before allocating the I/O pipe.* This means, for example, that the ports bound to `(current-input-port)` and `(current-output-port)` in either process are not affected—they still

43

refer to the same I/O sources and sinks as before. Remember the simple scsh rule: Scheme ports are bound to I/O sources and sinks, *not* particular file descriptors.

If the child process wishes to rebind the current output port to the pipe on file descriptor 1, it can do this using `with-current-output-port` or a related form. Similarly, if the parent wishes to change the current input port to the pipe on file descriptor 0, it can do this using `set-current-input-port!` or a related form. Here is an example showing how to set up the I/O ports on both sides of the pipe:

```
(fork/pipe (lambda ()
              (with-current-output-port (fdes->outport 1)
              (display "Hello, world.\\n"))))
  (set-current-input-port! (fdes->inport 0))
  (read-line)      ; Read the string output by the child.
```

None of this is necessary when the I/O is performed by an exec'd program in the child or parent process, only when the pipe will be referenced by Scheme code through one of the default current I/O ports.

```
(fork/pipe+ conns thunk [continue-threads?]) → (or/c proc? #f)
  conns : (listof integer?)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
(%fork/pipe+ conns thunk [continue-threads?]) → (or/c proc? #f)
  conns : (listof integer?)
  thunk : (or/c #f (-> (values any ...)))
  continue-threads? : boolean? = #f
```

Like `fork/pipe`, but the pipe connections between the child and parent are specified by the connection list `conns`. See the

```
("|+" conns pf1 ... pfn)
```

process form for a description of connection lists.

Index