9 10 10.95 12

An Executable Model of the Synergy File System

William R. Bevier
Richard M. Cohen

Computational Logic, Inc.
1717 West 6th Street, Suite 290
Austin, Texas 78703-4776

Telephone: 512 322 9951
Email: bevier@cli.com, cohen@cli.com

Technical Report 121 October, 1996

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

Contents

1 Introduction

This report presents the definitions and theorems that comprise an executable model of the Synergy File System specification. The model is written in ACL2 [KM94], and closely follows a specification for the interface written in Z [Spi89]. Unlike the Z specification, the ACL2 model can be tested. Since ACL2 includes a theorem prover, proofs of theorems about the model can be mechanically checked. The ability to execute and reason about a model provides a way to get increased assurance that one has specified desirable behavior.

The executable portion of ACL2, consisting of function definitions, macros, and global variables is very much like Common Lisp [Ste84, Ste90]. ACL2 contains some additional constructs, like `defthm` which make use of the automated theorem prover that is part of the ACL2 system.

To achieve executability in this model, we sometimes make implementation choices. We try to make these as innocuous as possible in terms of their effect on the behavior of the system. An example of such a choice is the use of numbers as file identifiers. Other data types might serve equally well.

This document closely follows the structure of the Z specification [BCT95]. After a brief introduction to ACL2 in Section 2, Section 4 lays out the central structure of the file system: the association of file identifiers and files. Section 5 adds directories and pathnames to this structure. Section 6 presents data strutures for opening and closing files. Section 7 introduces processes. Section 8 presents algorithms for garbage collection. Section 9 combines the previous sections into the first layer that resembles Unix file system functionality. Section 10 introduces discretionary access control. Finally, Section 11 models operations at the level of the Unix file system interface on successful outcomes. We do not model any of the many return codes.

The main benefit of writing this model is not clearly visible from reading a document such as this. One must run the model or prove theorems about it interactively to get the full impact. Doing this allows one to get a stronger impression as to whether what one has specified is what one wants.

Constructing this model caused us to find numerous errors in the Z specification. Some of these were simple typographical errors. Another class of problems was somewhat unique to Z, having to do with the way that schema inclusion matches variables with the same names (modulo decorations). At several places in the Z specification, we had failed to combine schemas correctly because certain names matched unintentionally, or because intended matches did not occur. These are hard to find by inspection. We found these errors in the careful implementation of the specification, building functions that take explicit parameters rather than relying on Z's name matching.

Simply building the ACL2 model led us to find some errors. More were found when we tested the model. Yet others were found when we attempted to prove some theorems about the model. As a result of this experience, we feel that while Z can be quite useful for conveying information precisely, it can be difficult to get the details right for a large specification. The executability of ACL2 allows one to construct and test a prototype implemenation with a modest amount of effort. Proving theorems about the resulting model can provide even more assurance, but at a potentially high cost for the effort put into the proofs.

The structure of the Z specification had a strong impact on the structure of the ACL2 model. In fact, we were able to closely follow the Z specification in the model. We believe that the model could be proved to satisfy the specification, but we have not done this.

Notes on Style. We present a functional model of the file system. In a C implementation, the state of the file system is accessed globally and is directly modified. In this model, operations on the system take a state parameter as argument, and return a resulting state. This allows us to more easily execute the model and reason about it, since we can keep track of a sequence of system states.

Functions are printed in the syntax of Common Lisp, not conventional mathematical notation. The conventional $\mathbf{x} + \mathbf{y}$ is displayed as `(+ x y)`. A defined function written conventionally as $\mathbf{fn}\,(\mathbf{x}, \mathbf{y})$ is displayed as `(fn x y)`.

ACL2 is an untyped language. In place of types we define functions in the logic that recognize an expression of a given type. Such a function is a *predicate*, and returns a boolean value. For example, instead of declaring `n` to be of type *number*, we write `(numberp n)` where appropriate.

2 A Brief Introduction to ACL2

In this section we summarize some of the features of ACL2. We rely to a large extent on the reader's assumed familiarity with Common Lisp.

Global Variables. The toplevel ACL2 form `assign` is used to assign a value to a global variable. The operator `@` returns the value of a global variable. For example, after executing `(assign x 3)`, the value of `(@ x)` is `3`.

Defun. The `defun` form creates a function. In the example below, the function `foo` is defined. `foo` has two arguments, `x` and `y`. Assumptions about the arguments are declared (optionally) in the guard. In this example, `x` and `y` are declared to be integers. The guard is evaluated at run time, and causes an error if it is not satisfied. Following the declaration is the body of function.

```
(defun foo (x y)
  (declare (xargs :guard (and (integerp x) (integerp y))))
  (* (+ x y) 2))
```

Multiple Values. A function may return more than one value. One way of returning multiple values is to return a list of values. However, using the ACL2 multiple value primitives, `mv` and `mv-let`, allows the system to check for the right number of values at the time a definition is processed. In this example, `dog` returns a multiple value, and `cat` uses an `mv-let` to unbind the values. `cat` returns the sum of a number and its double. `mv` and `mv-let` correspond to Common Lisp's `values` and `multiple-value-bind`.

```
(defun dog (x) (declare (xargs :guard (integerp x))) (mv x (* 2 x)))

(defun cat (y)
  (declare (xargs :guard (integerp y)))
  (mv-let (i j) (dog y) (+ i j)))
```

Defthm. A `defthm` form proposes a theorem about previously introduced functions. The mechanical proof checker within ACL2 attempts a proof of the proposed theorem. In this example, we suggest the theorem that the function `foo` returns an even number if its arguments are integers.

```
(defthm evenp-foo
        (implies (and (integerp x) (integerp y)) (evenp (foo x y))))
```

Deflist. `deflist` is a macro defined by the authors that generates a recursive function which recognizes a list, all of whose elements satisfy a given unary predicate. Additionally, it automatically generates a large number of `defthm` forms that inform the theorem prover of important properties of the new function. The following example introduces a function `integer-listp` that recognizes a list if integers.

```
(deflist integer-listp (l) integerp)
```

Defalist. `defalist` is a macro defined by the authors that generates a recognizer for a typed alist.[1] Theorems about accessing and constructing alists of the given type are automatically generated. The following form introduces a function `symbol->integer` that recognizes an alist which maps symbols to integers.

```
(defalist symbol->integer (l) (symbolp . integerp))
```

[1]An alist is short for *association list*. An alist can be used to associate a value with a key. Lisp provides the lookup function `assoc` to find the value associated with a key in an alist.

Defstructure. `defstructure` is a macro that provides a capability similar to Common Lisp's `defstruct`. It allows one to define a record structure, including its accessor, constructor and update functions. The following example defines a `person` record structure consisting of `height` and `weight` fields.

```
(defstructure person height weight)
```

The automatically generated functions include the following.

`(make-person :height h :weight w)`	construct a `person` structure
`(person-height p)`	access the `height` field of a `person`
`(person-weight p)`	access the `weight` field of a `person`
`(update-person p :weight w)`	update the `weight` field of a `person`
`(person-p p)`	a predicate that recognizes a `person` structure

Functions on lists. This is a brief synopsis of functions on lists. Some of these are primitive ACL2 functions, and some are defined by the authors.

`nil`	the empty list
`(consp x)`	`x` is a non-empty list
`(cons x list)`	the list whose first element is `x`, and whose remainder is `list`
`(car list)`	the first element of a non-empty list
`(cdr list)`	all but the first element of a non-empty list
`(append a b)`	the concatenation of two lists
`(nonlast list)`	all but the last element of `list`
`(member-equal x list)`	true if `x` is a member of `list`
`(nth-seg i j list)`	the sublist of `list` beginning at offset `i` of length `j`
`(put-seg-fill i seg list)`	replaces the sublist of `list` beginning at offset `i` with list `seg`

Functions on alists. This section provides a brief synopsis of functions on alists, all written by the authors.

`(binding key alist)`	gives the value associated with `key` in `alist`
`(bind key value alist)`	returns an update alist, with `key` bound to `value`
`(bound? key alist)`	true if `key` is bound to a value in `alist`
`(rembind key alist)`	returns an updated alist, with `key` unbound
`(domain alist)`	the list of keys in the alist
`(range alist)`	the list of values in the alist
`(all-bound? list alist)`	true when every key in `list` is bound in `alist`
`(inv-bound? value alist)`	`value` occurs in the range of `alist`
`(inv-binding value alist)`	the key to which `value` is bound in `alist`

3 Synergy File System Constants

This section contains constants that are arbitrary, but that must be defined for Synergy File Sytem model to be executable. `pathname-limit` is the maximum number of names in a path name.

```
(defun pathname-limit () 20)
```

4 The Basic File Space

4.1 The Model

This section models the basic data structure of the file space, a mapping from file identifiers (*fids*) to files. We model a byte as either a number or a literal. This allows us later to minimize the complexity of modeling directories.

```
(defun literalp (x)
  (and (symbolp x) (not (equal x t)) (not (equal x nil))))
```

```
(defun bytep (x) (or (naturalp x) (literalp x)))
```

An object is a file, *i.e.*, satisfies the predicate `filep` if it is a list of bytes. The predicate `file-listp` recognizes a list of files.

```
(deflist filep (l) bytep)
```

```
(deflist file-listp (l) filep)
```

We define the pad character for files to be `0`. The function `padfile` creates a file of length `n`, consisting entirely of pad characters.

```
(defun pad () 0)
```

```
(defun padfile (n)
  (declare (xargs :guard (naturalp n)))
  (make-list n :initial-element (pad)))
```

We model a file identifier as a natural number. We use the function `excess-natural` (defined by the authors) to compute an unused file identifier. The function `fid-listp` recognizes a list of file identifiers.

```
(defun fidp (x) (naturalp x))
```

```
(deflist fid-listp (l) fidp)
```

The predicate `fid->file` recognizes an alist that maps file identifiers to files. A data structure of this type is the central object to be managed by the file system. We call this data structure by the name `fcontents` throughout this document. `fcontents` satisfies the predicate `basic-file-space` if it is a mapping from fids to files.

```
(defalist fid->file (l) (fidp . filep))
```

```
(defun basic-file-space (fcontents) (fid->file fcontents))
```

The printed representation of an entry in `fcontents` appears as a list whose first element is the file identifier, and whose remainder is the contents of the file. Here is example containin two files, with identifiers 43 and 57. The contents of file 43 is `(a b c d e f g)`, and the contents of file 57 is `(w x y z)`.

```
'((43 a b c d e f g) (57 w x y z))
```

The function `newfid` returns a file identifier not currently in the domain of `fcontents`. The following theorem states that the function indeed returns an unused value. The function `filelength` gives the length of the file associated with an fid.[2]

```
(defun newfid (fcontents) (excess-natural (domain fcontents)))



(defthm member-newfid-domain-fcontents
        (implies (basic-file-space fcontents)
                 (not (member-equal (newfid fcontents)
                           (domain fcontents)))))


(defun filelength (fcontents fid?)
  (declare (xargs :guard
                  (and (basic-file-space fcontents) (fidp fid?)
                      (bound? fid? fcontents))))
  (length (binding fid? fcontents)))
```

The following functions model the four transitions on a basic file system state. `create-bfs` binds a new fid to an empty file. `destroy-bfs` removes a file identifier and its associated file from `fcontents`. `read-bfs` returns the contents of file `fid?` beginning at the given offset for the given length. `write-bfs` writes the sequence of bytes `data?` to the designated file at the given offset.

```
(defun create-bfs (fcontents fid?)
  (declare (xargs :guard
                  (and (basic-file-space fcontents) (fidp fid?)
                       (not (bound? fid? fcontents)))))
  (bind fid? nil fcontents))



(defun destroy-bfs (fcontents fid?)
  (declare (xargs :guard
                  (and (basic-file-space fcontents) (fidp fid?)
                       (bound? fid? fcontents))))
  (rembind fid? fcontents))
```

[2]We follow the convention of using variable names undecorated with punctuation to represent state variables. Variable names decorated with a question mark (?) are considered to represent input parameters.

```
(defun read-bfs (fcontents fid? offset? length?)
  (declare (xargs :guard
                  (and (basic-file-space fcontents) (fidp fid?)
                       (bound? fid? fcontents) (naturalp offset?)
                       (naturalp length?))))
  (nth-seg offset? length? (binding fid? fcontents)))
```

```
(defun write-bfs (fcontents fid? offset? data?)
  (declare (xargs :guard
                  (and (basic-file-space fcontents) (fidp fid?)
                       (bound? fid? fcontents) (naturalp offset?)
                       (filep data?))))
  (bind fid?
        (put-seg-fill offset? data? (binding fid? fcontents) (pad))
        fcontents))
```

4.2 Evaluation of the Model

Here is a sequence of basic file system operations. Let `fc0` be the initial, empty file space. `fc1` and `fc2` are the results of creating two empty files. In `fc3` we write to file 1 beginning at offset 0. In `fc4` we write to file 2 beginning at offset 4. Note that, following Unix file system semantics, pad characters are generated at the front. In `fc5` file 1 is destroyed.

```
(assign fc0 nil)

(assign fc1 (create-bfs (@ fc0) 1))

(equal (@ fc1) '((1)))

(assign fc2 (create-bfs (@ fc1) 2))

(equal (@ fc2) '((1) (2)))

(assign fc3 (write-bfs (@ fc2) 1 0 '(a b c d e f g h i j)))

(equal (@ fc3) '((1 a b c d e f g h i j) (2)))

(assign fc4 (write-bfs (@ fc3) 2 4 '(w x y z)))
```

```
(equal (@ fc4) '((1 a b c d e f g h i j) (2 0 0 0 0 w x y z)))

(assign fc5 (destroy-bfs (@ fc4) 1))

(equal (@ fc5) '((2 0 0 0 0 w x y z)))
```

The following example illustrates the execution of `read-bfs`. Reading file 2 from state `fc5` gives the following result.

```
(equal (read-bfs (@ fc5) 2 2 4) '(0 0 w x))
```

4.3 Theorems about the Model

The following theorems show that each of the basic file transitions preserve the basic file space invariant. That is, given an initial `fcontents` that satisfies `basic-file-space`, the result of the operation also satisfies this predicate.

```
(defthm basic-file-space-create-bfs
        (implies (and (basic-file-space fcontents) (fidp fid?))
                 (basic-file-space (create-bfs fcontents fid?))))

(defthm basic-file-space-destroy-bfs
        (implies (basic-file-space fcontents)
                 (basic-file-space (destroy-bfs fcontents fid?))))

(defthm basic-file-space-write-bfs
        (implies (and (basic-file-space fcontents) (fidp fid?)
                      (bound? fid? fcontents) (naturalp offset?)
                      (filep data?))
                 (basic-file-space
                     (write-bfs fcontents fid? offset? data?))))
```

5 The File Space

5.1 The Model

In this section we extend the basic file space model to incorporate directories and pathnames. A syllable is modeled as a literal. `syllable-listp` recognizes a list of syllables. A pathname is modeled as a syllable list.

```
(defun syllablep (x) (literalp x))
```

```
(deflist syllable-listp (l) syllablep)
```

```
(defun path-namep (l) (syllable-listp l))
```

A directory is modeled as an alist that maps syllables to fids. We store directories in the file system in "flattened" form. That is, a directory that has the following two entries `((mail . 1) (programs .2))` will be flattened into the list `(mail 1 programs 2)`, so that it will satisfy `filep`. This is accomplished by the function `flatten-alist` (not displayed here). The inverse operation is `make-alist`, which can construct a directory alist from a list of alternating syllables and file identifiers.

```
(defalist directoryp (l) (syllablep . fidp))
```

The following predicate recognizes a list of alternating syllables and file identifiers, i.e., a flattened directory.

```
(defun alternating-syllable-fid-listp (l)
  (cond
    ((atom l) (equal l nil))
    ((syllablep (car l))
     (and (not (atom (cdr l))) (fidp (cadr l))
          (alternating-syllable-fid-listp (cddr l))))
    (t nil)))
```

`parsedir` is our interface for parsing a flattened directory into a directory structure, and `unparsedir` is its inverse. The theorems below establish that, given appropriate inputs, these two functions produce results of the right type.

```
(defun parsedir (l)
  (declare (xargs :guard (alternating-syllable-fid-listp l)))
  (make-alist l))
```

```
(defun unparsedir (d)
  (declare (xargs :guard (directoryp d)))
  (flatten-alist d))
```

```
(defthm directoryp-parsedir
        (implies (alternating-syllable-fid-listp l)
                 (directoryp (parsedir l))))

(defthm alternating-syllable-fid-listp-unparsedir
        (implies (directoryp d)
                 (alternating-syllable-fid-listp (unparsedir d))))
```

We model two types of files: regular files and directories. The function `ftypep` recognizes a file type. The state variable `ftyp` is an alist that maps each file identifier to its type. The functions `add-ftype` and `delete-ftype` modify the file type alist by adding or removing an entry, respectively.

```
(defun ftypep (x) (member-equal x '(reg dir)))


(defalist fid->ftype (l) (fidp . ftypep))


(defun add-ftype (ftyp fid? ftyp?)
  (declare (xargs :guard
                  (and (fid->ftype ftyp) (fidp fid?) (ftypep ftyp?)
                       (not (bound? fid? ftyp)))))
  (bind fid? ftyp? ftyp))


(defun delete-ftype (ftyp fid?)
  (declare (xargs :guard
                  (and (fid->ftype ftyp) (fidp fid?)
                       (bound? fid? ftyp))))
  (rembind fid? ftyp))
```

The following predicates are used to state several important invariants on the file space. Let `lst` be a list of file identifiers. Then `(all-dirs-are-dir-files lst fcontents ftyp)` holds when every fid in `lst` whose type is `dir` is bound in `fcontents` to a file that is a flattened directory.

```
(defun all-dirs-are-dir-files (lst fcontents ftyp)
  (if (atom lst) t
      (let ((fid (car lst)))
        (and (if (equal (binding fid ftyp) 'dir)
                 (alternating-syllable-fid-listp
                     (binding fid fcontents))
                 t)
             (all-dirs-are-dir-files (cdr lst) fcontents ftyp)))))
```

Let `lst` be a list of file identifiers. Then `(all-dirs-map-to-legal-fids lst fcontents ftyp)` is true when every fid in `lst` whose type is `dir` is bound to a directory file that contains only fids that occur in `fcontents`. That is, no directory contains dangling file identifiers.

```
(defun all-dirs-map-to-legal-fids (lst fcontents ftyp)
  (if (atom lst) t
      (let ((fid (car lst)))
        (and (if (equal (binding fid ftyp) 'dir)
                 (all-bound? (range (parsedir (binding fid fcontents)))
                     fcontents)
                 t)
             (all-dirs-map-to-legal-fids (cdr lst) fcontents ftyp)))))
```

The following theorem captures the main consequence of `all-dirs-map-to-legal-fids`. If `fid` denotes a directory file bound in `fcontents`, then the file identifiers bound in that directory all occur in `fcontents`.

```
(defthm all-bound?-range-parsedir-binding-instance
        (implies (and (all-dirs-map-to-legal-fids (domain fcontents)
                          fcontents ftyp)
                      (equal (binding fid ftyp) 'dir)
                      (bound? fid fcontents) (alistp fcontents))
                 (all-bound? (range (parsedir (binding fid fcontents)))
                     fcontents)))
```

A file space consists of three state variables: `fcontents`, the assocation of file identifiers with files; `ftyp` the association of file identifiers with file types; and `rootfid`, the file identifier that is considered the root fid. We organize these state components into a record structure called `fs`, for *file space*.

The predicate `filespace` makes states explicit requirements on a legal file space. Each of the individual state variables has the right type. Additionally, the domains of `ftyp` and `fcontents` are equal. So, if an fid is bound in `ftyp`, then it's also bound in `fcontents`. The root fid occurs in the file space and is a directory file. All directory files are bound to flattened directories in the file space, and directory contains only existing fids.

```
(defstructure fs fcontents ftyp rootfid)
```

```
(defun filespace (fs)
  (let ((fcontents (fs-fcontents fs)) (ftyp (fs-ftyp fs))
        (rootfid (fs-rootfid fs)))
```

```
    (and (fs-p fs) (basic-file-space fcontents) (fid->ftype ftyp)
         (fidp rootfid) (equal (domain ftyp) (domain fcontents))
         (bound? rootfid fcontents) (equal (binding rootfid ftyp) 'dir)
         (all-dirs-are-dir-files (domain fcontents) fcontents ftyp)
         (all-dirs-map-to-legal-fids (domain fcontents) fcontents ftyp))))
```

The predicate `legal-pathname` recognizes a legal pathname `pn` relative to file identifier `fid` in a file space. That is, beginning at the given file identifier, the pathname gives a sequence of syllables that occur in successive directories. We have included some strong statements about the well-formedness of `fcontents` and `ftyp` within the definition of `legal-pathname`.

```
(defun legal-pathnamep (fid pn fcontents ftyp)
  (if (or (not (fidp fid)) (not (bound? fid ftyp))
          (not (bound? fid fcontents)))
      nil
      (if (atom pn) t
          (let* ((typ (binding fid ftyp))
                 (file (binding fid fcontents)) (dir (parsedir file)))
            (and (equal typ 'dir) (alternating-syllable-fid-listp file)
                 (bound? (car pn) dir)
                 (legal-pathnamep (binding (car pn) dir) (cdr pn)
                    fcontents ftyp))))))
```

The function `pathname-fid` gives the file identifier denoted by a legal pathname. In the rest of text of this report, we frequently refer to a pathname `pn` without mentioning the file identifier to which it is relative. In the ACL2 forms, however, a starting file identifier `startfid` is made explicit.

```
(defun pathname-fid (fid pn fcontents)
  (if (atom pn) fid
      (pathname-fid
          (binding (car pn) (parsedir (binding fid fcontents)))
          (cdr pn) fcontents)))
```

The first of the following functions recognizes a pathname that denotes a regular file. The second function recognizes a pathname that denotes an empty directory.

```
(defun regular-file (fcontents ftyp startfid pn)
  (declare (xargs :guard (legal-pathnamep startfid pn fcontents ftyp)))
  (equal (binding (pathname-fid startfid pn fcontents) ftyp) 'reg))
```

```
(defun empty-directory (fcontents ftyp startfid pn)
  (declare (xargs :guard (legal-pathnamep startfid pn fcontents ftyp)))
  (and (equal (binding (pathname-fid startfid pn fcontents) ftyp) 'dir)
       (equal (binding (pathname-fid startfid pn fcontents) fcontents)
              nil)))
```

The following functions define transitions on the file space state. `link-fs` creates a new link to a file. `pn?` is a non-empty pathname, all but the last element of which identifies a directory in the file space. The last element of `pn?` is an unused syllable in that directory. `link-fs` associates that syllable with `fid?` in that directory, creating a new link to `fid?`.

```
(defun link-fs (fs startfid? pn? fid?)
  (declare (xargs :guard
                  (let* ((fcontents (fs-fcontents fs))
                         (ftyp (fs-ftyp fs))
                         (fid (pathname-fid startfid? (nonlast pn?)
                                   fcontents)))
                    (and (filespace fs) (bound? fid? ftyp)
                         (path-namep pn?) (< 0 (len pn?))
                         (legal-pathnamep startfid? (nonlast pn?)
                             fcontents ftyp)
                         (equal (binding fid ftyp) 'dir)
                         (not (bound? (car (last pn?))
                                      (parsedir
                                       (binding fid fcontents)))))))))
  (let* ((fcontents (fs-fcontents fs))
         (parentfid (pathname-fid startfid? (nonlast pn?) fcontents))
         (newdir (bind (car (last pn?)) fid?
                       (parsedir (binding parentfid fcontents)))))
    (update-fs fs :fcontents
        (bind parentfid (unparsedir newdir) fcontents))))
```

`unlink-fs` removes the last syllable in pathname `pn?` from its parent directory.

```
(defun unlink-fs (fs startfid? pn?)
  (declare (xargs :guard
                  (let ((fcontents (fs-fcontents fs))
                        (ftyp (fs-ftyp fs)))
                    (and (filespace fs) (< 0 (len pn?))
                         (legal-pathnamep startfid? pn? fcontents ftyp)))))
  (let* ((fcontents (fs-fcontents fs))
         (parentfid (pathname-fid startfid? (nonlast pn?) fcontents))
         (newdir (rembind (car (last pn?))
                        (parsedir (binding parentfid fcontents)))))
    (update-fs fs :fcontents
        (bind parentfid (unparsedir newdir) fcontents))))
```

`create-fs` applies the `create-bfs` transition to make a new, empty file. Then, that file is linked to the pathname.

```
(defun create-fs (fs startfid? pn? ftyp?)
  (declare (xargs :guard
                 (let* ((fcontents (fs-fcontents fs))
                        (ftyp (fs-ftyp fs))
                        (fid (pathname-fid startfid? (nonlast pn?)
                                 fcontents)))
                   (and (filespace fs) (path-namep pn?)
                        (< 0 (len pn?))
                        (legal-pathnamep startfid? (nonlast pn?)
                            fcontents ftyp)
                        (equal (binding fid ftyp) 'dir)
                        (not (bound? (car (last pn?))
                                     (parsedir
                                      (binding fid fcontents))))
                        (ftypep ftyp?)))))
  (let* ((fcontents (fs-fcontents fs)) (ftyp (fs-ftyp fs))
         (newfid (newfid fcontents))
         (fs* (update-fs fs :fcontents (create-bfs fcontents newfid)
                 :ftyp (add-ftype ftyp newfid ftyp?))))
    (link-fs fs* startfid? pn? newfid)))
```

5.2 Evaluation of the Model

Here is a sequence of file system operations. `fs0` is an initial file system state, with `rootfid = 0`. The rootfid is bound to an empty file in `fcontents`, and to the value `directory` in `ftyp`. In the first step, we create the subdirectory `usr` of root. Then the subdirectory `tmp` is created. Next, the subdirectory `smith` of `usr` is created. Finally, `smith` is unlinked. Evaluating the function `filespace`, the legal state predicate, on the result of each of these steps results in the value `t` (*true*).

```
(assign fs0
        (make-fs :fcontents (bind 0 nil nil) :ftyp (bind 0 'dir nil)
                 :rootfid 0))
```

```
(equal (@ fs0) '(fs ((0)) ((0 . dir)) 0))

(assign fs1 (create-fs (@ fs0) 0 '(usr) 'dir))

(equal (@ fs1) '(fs ((0 usr 1) (1)) ((0 . dir) (1 . dir)) 0))

(assign fs2 (create-fs (@ fs1) 0 '(tmp) 'dir))

(equal (@ fs2)
       '(fs ((0 usr 1 tmp 2) (1) (2)) ((0 . dir) (1 . dir) (2 . dir))
            0))

(assign fs3 (create-fs (@ fs2) 0 '(usr smith) 'dir))

(equal (@ fs3)
       '(fs ((0 usr 1 tmp 2) (1 smith 3) (2) (3))
            ((0 . dir) (1 . dir) (2 . dir) (3 . dir)) 0))

(assign fs4 (unlink-fs (@ fs3) 1 '(smith)))

(equal (@ fs4)
       '(fs ((0 usr 1 tmp 2) (1) (2) (3))
            ((0 . dir) (1 . dir) (2 . dir) (3 . dir)) 0))
```

5.3 Theorems about the Model

The following theorems show that the `link-fs` and `unlink-fs` transitions preserve the file space invariant. We have not proved the theorem for `create-fs`.

```
(defthm filespace-link-fs
        (let* ((fcontents (fs-fcontents fs)) (ftyp (fs-ftyp fs))
               (fid (pathname-fid startfid? (nonlast pn?) fcontents)))
          (implies (and (filespace fs) (bound? fid? ftyp)
                        (path-namep pn?) (< 0 (len pn?))
                        (legal-pathnamep startfid? (nonlast pn?)
                           fcontents ftyp)
                        (equal (binding fid ftyp) 'dir)
                        (not (bound? (car (last pn?))
                                     (parsedir (binding fid fcontents)))))
                   (filespace (link-fs fs startfid? pn? fid?)))))

(defthm filespace-unlink-fs
        (let ((fcontents (fs-fcontents fs)) (ftyp (fs-ftyp fs)))
          (implies (and (filespace fs) (< 0 (len pn?))
                        (legal-pathnamep startfid? pn? fcontents ftyp))
                   (filespace (unlink-fs fs startfid? pn?)))))
```

6 The File Table

6.1 The Model

The file table records information about open files. An `oid` is an identifier for an entry in the file table. We define oids to be natural numbers.

```
(defun oidp (x) (naturalp x))
```

```
(deflist oid-listp (l) oidp)
```

The function `modep` recognizes a legal file access mode.

```
(defun modep (x) (member-equal x '(rdonly wronly rdwr append)))
```

```
(deflist mode-listp (l) modep)
```

The file table consists of three state variables. `posn` is a mapping from oids to natural numbers, recording the current offset into an open file. `mode` is a mapping from oid to access mode, recording the mode in which a file is open. `fid` is a mapping from oid to fid, recording the file that is open under a given oid. We organize these state components into a record structure called `ft`, for *file table.*

A legal file table is an `ft` structure in which all three of the components are of the right type. The domains of each of the three components must be equal. That is, when an oid occurs in one, it occurs in all.

```
(defstructure ft posn mode fid)
```

```
(defun filetable (ft)
  (let ((ftposn (ft-posn ft)) (ftmode (ft-mode ft))
        (ftfid (ft-fid ft)))
    (and (ft-p ft) (oid->natural ftposn) (oid->mode ftmode)
         (oid->fid ftfid) (equal (domain ftposn) (domain ftmode))
         (equal (domain ftmode) (domain ftfid)))))
```

`new-oid` returns an unused oid.

```
(defun new-oid (alist) (excess-natural (domain alist)))
```

The following functions define transitions on the file table. `open-ft` creates a new oid and associates with it a file position, mode, and file identifier. `close-ft` removes an oid from the file table. `seek-ft` updates an oid's position. `incrposn-ft` increments an oid's position.

```
(defun open-ft (ft posn? mode? fid?)
  (declare (xargs :guard
                  (and (filetable ft) (naturalp posn?) (modep mode?)
                       (fidp fid?))))
  (let* ((ftposn (ft-posn ft)) (ftmode (ft-mode ft))
         (ftfid (ft-fid ft)) (newoid (new-oid ftfid)))
    (mv newoid
        (update-ft ft :posn (bind newoid posn? ftposn) :mode
            (bind newoid mode? ftmode) :fid (bind newoid fid? ftfid)))))
```

```
(defun close-ft (ft oid?)
  (declare (xargs :guard
                  (and (filetable ft) (bound? oid? (ft-fid ft)))))
  (let ((ftposn (ft-posn ft)) (ftmode (ft-mode ft))
        (ftfid (ft-fid ft)))
    (update-ft ft :posn (rembind oid? ftposn) :mode
        (rembind oid? ftmode) :fid (rembind oid? ftfid))))
```

```
(defun seek-ft (ft oid? offset?)
  (declare (xargs :guard
                  (and (filetable ft) (bound? oid? (ft-fid ft))
                       (naturalp offset?))))
  (update-ft ft :posn (bind oid? offset? (ft-posn ft))))
```

```
(defun incrposn-ft (ft oid? delta?)
  (declare (xargs :guard
                  (and (filetable ft) (bound? oid? (ft-fid ft))
                       (naturalp delta?))))
  (let ((ftposn (ft-posn ft)))
    (update-ft ft :posn
        (bind oid? (+ (binding oid? ftposn) delta?) ftposn))))
```

The following are useful predicates on the file table. Each checks whether an oid represents a file opened in a given mode.

```
(defun opened-for-read (ft oid?)
  (declare (xargs :guard
                  (and (filetable ft) (oidp oid?)
                       (bound? oid? (ft-posn ft)))))
  (member-equal (binding oid? (ft-mode ft)) '(rdonly rdwr)))


(defun opened-for-write (ft oid?)
  (declare (xargs :guard
                  (and (filetable ft) (oidp oid?)
                       (bound? oid? (ft-posn ft)))))
  (member-equal (binding oid? (ft-mode ft)) '(wronly rdwr)))


(defun opened-for-append (ft oid?)
  (declare (xargs :guard
                  (and (filetable ft) (oidp oid?)
                       (bound? oid? (ft-posn ft)))))
  (equal (binding oid? (ft-mode ft)) 'append))
```

6.2 Evaluation of the Model

Here are some forms illustrating execution of the file table primitives. Two files are opened, the position associated with the second file is updated. Then the first file is closed. Note that these transitions are defined independently of the file space.

```
(assign ft0 (make-ft :posn nil :mode nil :fid nil))

(equal (@ ft0) '(ft nil nil nil))

(mv-let (oid ft*) (open-ft (@ ft0) 0 'rdonly 12)
        (progn (assign newoid oid) (assign ft1 ft*)))

(equal (@ ft1) '(ft ((0 . 0)) ((0 . rdonly)) ((0 . 12))))

(mv-let (oid ft*) (open-ft (@ ft1) 0 'rdwr 13)
        (progn (assign newoid oid) (assign ft2 ft*)))
```

```
(equal (@ ft2)
       '(ft ((0 . 0) (1 . 0)) ((0 . rdonly) (1 . rdwr))
            ((0 . 12) (1 . 13))))

(assign ft3 (seek-ft (@ ft2) 1 64))

(equal (@ ft3)
       '(ft ((0 . 0) (1 . 64)) ((0 . rdonly) (1 . rdwr))
            ((0 . 12) (1 . 13))))

(assign ft4 (close-ft (@ ft3) 0))

(equal (@ ft4) '(ft ((1 . 64)) ((1 . rdwr)) ((1 . 13))))
```

6.3 Theorems about the Model

For two of the transitions, we prove that the file table invariant is preserved, assuming that the arguments to the transitions satisfy the appropriate guards.

```
(defthm filetable-open-ft
        (implies (and (filetable ft) (naturalp posn?) (modep mode?)
                      (fidp fid?))
                 (mv-let (newoid ft*) (open-ft ft posn? mode? fid?)
                         (declare (ignore newoid)) (filetable ft*))))

(defthm filetable-close-ft
        (implies (and (filetable ft) (bound? oid? (ft-fid ft)))
                 (filetable (close-ft ft oid?))))
```

7 Processes

7.1 The Model

A system is populated by processes. A process identifier is recognized by the predicate `pidp`. As with the other types of identifiers, we have chosen to use natural numbers to represent process identifiers. `pid-listp` recognizes a list of process identifiers.

```
(defun pidp (x) (naturalp x))
```

```
(deflist pid-listp (l) pidp)
```

Processes access open files through *file descriptors.* A file descriptor is recognized by the predicate `fdp`. `fd-listp` recognizes a list of file descriptors.

```
(defun fdp (x) (naturalp x))
```

```
(deflist fd-listp (l) fdp)
```

The key data structure associated with processes is the process table. Each process is associated with an alist that maps its file descriptors to oids. We model this association with an alist that maps each process id to an alist which in turn maps that process's fd's to oid's. The predicate `process-table` recognizes a well-formed process table.

```
(defalist fd->oid (l) (fdp . oidp))
```

```
(deflist fd->oid-listp (l) fd->oid)
```

```
(defalist pid->[fd->oid] (l) (pidp . fd->oid))
```

```
(defun process-table (ptable) (pid->[fd->oid] ptable))
```

The function `fd-to-oid` maps a pid and fd to an oid.

```
(defun fd-to-oid (ptable pid? fd?)
  (declare (xargs :guard
                  (and (process-table ptable) (pidp pid?) (fdp fd?)
                       (bound? pid? ptable)
                       (bound? fd? (binding pid? ptable)))))
  (binding fd? (binding pid? ptable)))
```

Here are the transitions defined on the process table. `add-process-fd` adds a new fd to an existing process in the process table. `delete-process-fd` removes an fd from a process. `fork-pt` copies a process's fds to a newly created process.

```
(defun add-process-fd (ptable pid? oid?)
  (declare (xargs :guard
                  (and (process-table ptable) (pidp pid?) (oidp oid?)
                       (bound? pid? ptable))))
  (let ((newfd (excess-natural (domain (binding pid? ptable)))))
    (mv newfd
        (bind pid? (bind newfd oid? (binding pid? ptable)) ptable))))
```

```
(defun delete-process-fd (ptable pid? fd?)
  (declare (xargs :guard
                  (and (process-table ptable) (pidp pid?) (fdp fd?)
                       (bound? pid? ptable)
                       (bound? fd? (binding pid? ptable)))))
  (bind pid? (rembind fd? (binding pid? ptable)) ptable))
```

```
(defun fork-pt (ptable pid?)
  (declare (xargs :guard
                  (and (process-table ptable) (pidp pid?)
                       (bound? pid? ptable))))
  (let ((newpid (excess-natural (domain ptable))))
    (mv newpid (bind newpid (binding pid? ptable) ptable))))
```

7.2 Evaluation of the Model

This section illustrates a few transitions on the process table. The initil process table `pt0` contains two processes, 100 and 200. Neither have any fds. In the first step, a new fs is created for process 100, and associated with oid 1. In the next step, process gets another fd, associated with oid 2. Finally, fd 0 is removed from process 100's file descriptor table.

```
(assign pt0 (list (cons 100 nil) (cons 200 nil)))

(equal (@ pt0) '((100) (200)))

(mv-let (newfd newpt) (add-process-fd (@ pt0) 100 1)
        (declare (ignore newfd)) (assign pt1 newpt))

(equal (@ pt1) '((100 (0 . 1)) (200)))
```

```
(mv-let (newfd newpt) (add-process-fd (@ pt1) 100 2)
        (declare (ignore newfd)) (assign pt2 newpt))

(equal (@ pt2) '((100 (0 . 1) (1 . 2)) (200)))

(assign pt3 (delete-process-fd (@ pt2) 100 0))

(equal (@ pt3) '((100 (1 . 2)) (200)))
```

8 Garbage Collection

This section implements the garbage collection algorithms specified in [BCT95]. For the sake of brevity, we have chosen to omit display of some of the supporting functions in this part of the script.

`exists-reference-to-oid` asks if some process in the process table contains a file descriptor that maps to a given oid. The supporting function `exists-oid-reference` is not displayed in this document.

```
(defun exists-reference-to-oid (ptable oid?)
  (declare (xargs :guard (and (process-table ptable) (oidp oid?))))
  (exists-oid-reference (domain ptable) oid? ptable))
```

`exists-link-reference-to-fid` asks whether there is any pathname to a given fid in a file space. The function `all-pathnames`, not displayed here, computes the set of pathnames rooted at the system root fid.

```
(defun exists-link-reference-to-fid (fs fid?)
  (declare (xargs :guard (and (filespace fs) (fidp fid?))))
  (let ((fcontents (fs-fcontents fs)) (ftyp (fs-ftyp fs))
        (rootfid (fs-rootfid fs)))
    (exists-pathname-to-fid rootfid
        (all-pathnames rootfid fcontents ftyp) fid? fcontents ftyp)))
```

There exists a reference to a file identifier `fid` in the file table if some oid is mapped to `fid`.

```
(defun exists-file-table-reference-to-fid (ft fid?)
  (declare (xargs :guard (and (filetable ft) (fidp fid?))))
  (exists-file-table-reference (domain (ft-fid ft)) fid? (ft-fid ft)))
```

There exists a reference to a file identifier in the system if there is a pathname to it or if it is open.

```
(defun exists-reference-to-fid (fs ft fid?)
  (declare (xargs :guard
                  (and (filespace fs) (filetable ft) (fidp fid?))))
  (or (exists-link-reference-to-fid fs fid?)
      (exists-file-table-reference-to-fid ft fid?)))
```

File space garbage collection occurs with respect to a given file identifier if there are no references to it.

```
(defun file-space-gc (fs ft fid?)
  (declare (xargs :guard
                  (and (filespace fs) (filetable ft) (fidp fid?))))
  (if (exists-reference-to-fid fs ft fid?) fs
      (update-fs fs :fcontents (destroy-bfs (fs-fcontents fs) fid?)
          :ftyp (delete-ftype (fs-ftyp fs) fid?))))
```

An oid is garbage collected in the file table if there are no references to it.

```
(defun file-table-gc (ft pt oid?)
  (declare (xargs :guard
                  (and (filetable ft) (process-table pt) (oidp oid?))))
  (if (exists-reference-to-oid pt oid?) ft (close-ft ft oid?)))
```

9 The Basic Access System

9.1 The Model

This level of the model combines the file space, the file table, and the process table. The transitions at this level closely follow the semantics of Unix file system transitions, except that permissions and error conditions are not yet in the picture.

The structure of the basic access system (BAS) includes three components: the file space, the file table and the process table. `basic-access-system` defines a legal BAS structure. The requirements on the argument `bas` are

1. `bas` satisfies `bas-p`, which means that it is a `bas` defstructure.

2. The file space is legal.
3. The file table is legal.
4. The process table is legal.
5. Every oid in the file table maps to an fid found in the filespace.
6. The set of oids found in the process table (`(union$ (map-range (range ptable)))`) equals the set of oids found in the file table (`(domain ftfid)`).
7. Every pathname has a link. This, in effect, commits an implementation to perform garbage collection

```
(defstructure bas fspace ftable ptable)


(defun basic-access-system (bas)
  (let ((fcontents (fs-fcontents (bas-fspace bas)))
        (ftyp (fs-ftyp (bas-fspace bas)))
        (rootfid (fs-rootfid (bas-fspace bas)))
        (ftfid (ft-fid (bas-ftable bas))) (ptable (bas-ptable bas)))
    (and (bas-p bas) (filespace (bas-fspace bas))
         (filetable (bas-ftable bas)) (process-table (bas-ptable bas))
         (subsetp-equal (range ftfid) (domain fcontents))
         (set-equal (union$ (map-range (range ptable))) (domain ftfid))
         (exists-pathname-to-all-fids rootfid
             (all-pathnames rootfid fcontents ftyp) (domain fcontents)
             fcontents ftyp))))
```

Here are the BAS level transitions. The arguments to these transitions more closely resemble those at the Unix file system interface. `open-bas` opens the file linked to pathname `pn?` in mode `mode?`. It returns two values: a new file descriptor and an updated `bas` state.

```
(defun open-bas (bas pid? startfid? pn? mode?)
  (declare (xargs :guard
                  (let* ((fcontents (fs-fcontents (bas-fspace bas)))
                         (ptable (bas-ptable bas))
                         (ftyp (fs-ftyp (bas-fspace bas)))
                         (fid (pathname-fid startfid? pn? fcontents)))
                    (and (basic-access-system bas) (pidp pid?)
                         (bound? pid? ptable) (fidp startfid?)
                         (path-namep pn?)
                         (legal-pathnamep startfid? pn? fcontents ftyp)
```

```
                       (modep mode?)
                       (implies (equal (binding fid ftyp) 'dir)
                                (equal mode? 'rdonly))))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (pt (bas-ptable bas)) (fcontents (fs-fcontents fs))
         (fid (pathname-fid startfid? pn? fcontents))
         (fcontents*
             (case mode?
               ((wronly rdwr) (create-bfs fcontents fid))
               (t fcontents)))
         (newposn (if (equal mode? 'append)
                      (len (binding fid fcontents)) 0)))
    (mv-let (oid ft*) (open-ft ft newposn mode? fid)
            (mv-let (fd pt*) (add-process-fd pt pid? oid)
                    (mv fd
                        (update-bas bas :fspace
                            (update-fs (bas-fspace bas) :fcontents
                                fcontents*)
                            :ftable ft* :ptable pt*))))))
```

`close-bas` closes the file that process `pid?` accesses through file descriptor `fd?`.

```
(defun close-bas (bas pid? fd?)
  (declare (xargs :guard
                  (let ((pt (bas-ptable bas)))
                    (and (basic-access-system bas) (pidp pid?)
                         (bound? pid? pt) (fdp fd?)
                         (bound? fd? (binding pid? pt))))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (pt (bas-ptable bas)) (oid (binding fd? (binding pid? pt)))
         (pt* (delete-process-fd pt pid? fd?))
         (ft* (file-table-gc ft pt* oid))
         (fs* (file-space-gc fs ft* (binding oid (ft-fid ft)))))
    (update-bas bas :fspace fs* :ftable ft* :ptable pt*)))
```

`link-bas` makes creates a link to a file. As in `link-fs`, `pn?` is a non-empty pathname, all but the last element of which identifies a directory in the file space. The last element of `pn?` is an unused syllable in that directory. `link-bas` associates that syllable with the file linked to `xpn?`, the target pathname.

```
(defun link-bas (bas startfid? pn? xstartfid? xpn?)
  (declare (xargs :guard
                  (let ((fcontents (fs-fcontents (bas-fspace bas)))
                        (ftyp (fs-ftyp (bas-fspace bas))))
                    (and (basic-access-system bas) (fidp startfid?)
                         (path-namep pn?) (fidp xstartfid?)
                         (path-namep xpn?)
                         (legal-pathnamep xstartfid? xpn? fcontents
                             ftyp)))))
  (let* ((fcontents (fs-fcontents (bas-fspace bas)))
         (fid (pathname-fid xstartfid? xpn? fcontents))
         (fs* (link-fs (bas-fspace bas) startfid? pn? fid)))
    (update-bas bas :fspace fs*)))
```

`unlink-bas` unlinks the pathname `pn?` from its target file. Any resulting garbage is collected from the file space.

```
(defun unlink-bas (bas startfid? pn?)
  (declare (xargs :guard
                  (let ((fcontents (fs-fcontents (bas-fspace bas)))
                        (ftyp (fs-ftyp (bas-fspace bas))))
                    (and (basic-access-system bas) (fidp startfid?)
                         (path-namep pn?)
                         (legal-pathnamep startfid? pn? fcontents ftyp)
                         (or (regular-file fcontents ftyp startfid?
                                 pn?)
                             (empty-directory fcontents ftyp startfid?
                                 pn?))))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (fcontents (fs-fcontents fs))
         (fid (pathname-fid startfid? pn? fcontents))
         (fs* (unlink-fs fs startfid? pn?))
         (fs** (file-space-gc fs* ft fid)))
    (update-bas bas :fspace fs**)))
```

`create-bas` creates a new file in the filespace. Its type is given by `ftyp?`.

```
(defun create-bas (bas startfid? pn? ftyp?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (fidp startfid?)
                       (path-namep pn?) (ftypep ftyp?))))
  (let* ((fs (bas-fspace bas))
         (fs* (create-fs fs startfid? pn? ftyp?)))
    (update-bas bas :fspace fs*)))
```

`read-bas` performs a read operation in the file space. The file is identified by an file desscriptor `fd?` resulting from a previous open operation. Two values are returned: the sequence of bytes read, and an updated `bas` state. The state is updated to reflect the new position in the open file.

```
(defun read-bas (bas pid? fd? length?)
  (declare (xargs :guard
                  (let* ((ft (bas-ftable bas)) (pt (bas-ptable bas))
                         (oid (fd-to-oid pt pid? fd?)))
                    (and (basic-access-system bas) (pidp pid?)
                         (bound? pid? pt) (fdp fd?)
                         (bound? fd? (binding pid? pt))
                         (naturalp length?) (opened-for-read ft oid)))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (pt (bas-ptable bas)) (oid (fd-to-oid pt pid? fd?))
         (fid (binding oid (ft-fid ft)))
         (posn (binding oid (ft-posn ft)))
         (data (read-bfs (fs-fcontents fs) fid posn length?))
         (ft* (incrposn-ft ft oid (length data))))
    (mv data (update-bas bas :ftable ft*))))
```

`write-bas` writes the sequence of bytes `data?` at the current position of the file specified by file descriptor `fd?`.

```
(defun write-bas (bas pid? fd? data?)
  (declare (xargs :guard
                  (let* ((ft (bas-ftable bas)) (pt (bas-ptable bas))
                         (oid (fd-to-oid pt pid? fd?)))
                    (and (basic-access-system bas) (pidp pid?)
                         (bound? pid? pt) (fdp fd?)
                         (bound? fd? (binding pid? pt)) (filep data?)
                         (or (opened-for-write ft oid)
                             (opened-for-append ft oid))))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (pt (bas-ptable bas)) (oid (fd-to-oid pt pid? fd?))
         (fid (binding oid (ft-fid ft)))
         (posn (binding oid (ft-posn ft)))
         (fcontents* (write-bfs (fs-fcontents fs) fid posn data?))
         (ft* (incrposn-ft ft oid (length data?))))
    (update-bas bas :fspace (update-fs fs :fcontents fcontents*)
        :ftable ft*)))
```

`append-bas` writes the sequence of bytes `data?` at end of the file specified by file descriptor `fd?`.

```
(defun append-bas (bas pid? fd? data?)
  (declare (xargs :guard
                  (let* ((ft (bas-ftable bas)) (pt (bas-ptable bas))
                         (oid (fd-to-oid pt pid? fd?)))
                    (and (basic-access-system bas) (pidp pid?)
                         (bound? pid? pt) (fdp fd?)
                         (bound? fd? (binding pid? pt)) (filep data?)
                         (or (opened-for-write ft oid)
                             (opened-for-append ft oid))))))
  (let* ((fs (bas-fspace bas)) (ft (bas-ftable bas))
         (pt (bas-ptable bas)) (oid (fd-to-oid pt pid? fd?))
         (ft* (seek-ft ft oid
                       (filelength (fs-fcontents fs)
                          (binding oid (ft-fid ft))))))
    (write-bas (update-bas bas :ftable ft*) pid? fd? data?)))
```

`fork-bas` creates a new process, and copies the process table of the parent process into the new process. A new process identifier and an updates `bas` state are returned.

```
(defun fork-bas (bas pid?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (pidp pid?)
                       (bound? pid? (bas-ptable bas)))))
  (mv-let (pid pt*) (fork-pt (bas-ptable bas) pid?)
          (mv pid (update-bas bas :ptable pt*))))
```

9.2 Evaluation of the Model

In this section we display a fairly extensive sequence of calls to the BAS system interface. This sequence illustrates file creation, open and close operations, and reading and writing. Closing and unlinking files demonstrates garbage collection.

The initial BAS state is constructed from the initial file space, initial file table, and initial process table introduced in previous sections. The file space contains only the root fid. The file table is empty. The process table contains two process identifiers: 100 and 200, neither of which have any open files.

```
(assign bas0 (bas (@ fs0) (@ ft0) (@ pt0)))
```

```
(equal (@ bas0)
       '(bas (fs ((0)) ((0 . dir)) 0) (ft nil nil nil) ((100) (200))))
```

In the next sequence of commands, three subdirectories of `/` are created: `usr`, `tmp`, and `dev`. Then, the subdirectories `/usr/jones` and `/usr/smith` are created. Finally, the regular file `/usr/jones/mail` is created.

```
(assign bas1 (create-bas (@ bas0) 0 '(usr) 'dir))

(equal (@ bas1)
       '(bas (fs ((0 usr 1) (1)) ((0 . dir) (1 . dir)) 0)
             (ft nil nil nil) ((100) (200))))

(assign bas2 (create-bas (@ bas1) 0 '(tmp) 'dir))

(equal (@ bas2)
       '(bas (fs ((0 usr 1 tmp 2) (1) (2))
                 ((0 . dir) (1 . dir) (2 . dir)) 0)
             (ft nil nil nil) ((100) (200))))

(assign bas3 (create-bas (@ bas2) 0 '(dev) 'dir))

(equal (@ bas3)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1) (2) (3))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir)) 0)
             (ft nil nil nil) ((100) (200))))

(assign bas4 (create-bas (@ bas3) 0 '(usr jones) 'dir))

(equal (@ bas4)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4) (2) (3) (4))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)) 0)
             (ft nil nil nil) ((100) (200))))

(assign bas5 (create-bas (@ bas4) 1 '(smith) 'dir))

(equal (@ bas5)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3) (4)
                  (5))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir))
                 0)
             (ft nil nil nil) ((100) (200))))
```

```
(assign bas6 (create-bas (@ bas5) 0 '(usr jones mail) 'reg))

(equal (@ bas6)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft nil nil nil) ((100) (200))))
```

In the following sequence, `/usr/jones/mail` is opened in append mode. Two messages are appended, then the file is closed. Note that the file table is cleaned up after the file is closed. This is done via garbage collection.

```
(mv-let (fd bas) (open-bas (@ bas6) 100 0 '(usr jones mail) 'append)
        (progn (assign fd0 fd) (assign bas7 bas)))

(equal (@ bas7)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 0)) ((0 . append)) ((0 . 6)))
             ((100 (0 . 0)) (200))))

(assign bas8 (append-bas (@ bas7) 100 (@ fd0) '(a b c d e f g h)))

(equal (@ bas8)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 8)) ((0 . append)) ((0 . 6)))
             ((100 (0 . 0)) (200))))

(assign bas9 (append-bas (@ bas8) 100 (@ fd0) '(i j k l m n o p)))
```

```
(equal (@ bas9)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 16)) ((0 . append)) ((0 . 6)))
             ((100 (0 . 0)) (200))))
```

```
(assign bas10 (close-bas (@ bas9) 100 (@ fd0)))
```

```
(equal (@ bas10)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft nil nil nil) ((100) (200))))
```

Next, `/usr/jones/mail` is opened for reading. Two read operations are performed. Note how the file position is updated after each read. In the second read operation, the position is updated only by the amount read, not by the amount requested. Finally, the file is closed and the file table is gc'd.

```
(mv-let (fd bas) (open-bas (@ bas10) 200 0 '(usr jones mail) 'rdonly)
        (progn (assign fd1 fd) (assign bas11 bas)))
```

```
(equal (@ bas11)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 0)) ((0 . rdonly)) ((0 . 6)))
             ((100) (200 (0 . 0)))))
```

```
(mv-let (data bas) (read-bas (@ bas11) 200 (@ fd1) 5)
        (progn (assign data0 data) (assign bas12 bas)))
```

```
(equal (@ data0) '(a b c d e))
```

```
(equal (@ bas12)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 5)) ((0 . rdonly)) ((0 . 6)))
             ((100) (200 (0 . 0)))))

(mv-let (data bas) (read-bas (@ bas12) 200 (@ fd1) 20)
        (progn (assign data1 data) (assign bas13 bas)))

(equal (@ data1) '(f g h i j k l m n o p))

(equal (@ bas13)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft ((0 . 16)) ((0 . rdonly)) ((0 . 6)))
             ((100) (200 (0 . 0)))))

(assign bas14 (close-bas (@ bas13) 200 (@ fd1)))

(equal (@ bas14)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft nil nil nil) ((100) (200))))
```

`/usr/jones/mail` is not open. So unlinking the pathname to this file also removes it from the file space.

```
(assign bas15 (unlink-bas (@ bas14) 0 '(usr jones mail)))

(equal (@ bas15)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3) (4)
                  (5))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir))
                 0)
             (ft nil nil nil) ((100) (200))))
```

10 Discretionary Access Control

In this section we model Unix file system discretionary access control. A permission is one of {`r`, `w`, `x`}. User names and group names are symbols.

```
(defun permp (x) (declare (xargs :guard t)) (member-equal x '(r w x)))
```

```
(deflist perm-listp (l) permp)
```

```
(defun userp (x) (symbolp x))
```

```
(deflist user-listp (l) userp)
```

```
(defun groupp (x) (symbolp x))
```

```
(deflist group-listp (l) groupp)
```

The structure `dac-fa` contains the attributes that can be associated with a file: owner, group, owner permissions group permissions, and other permissions. `fid->dac-fa` models the association of a file identifier with the file's attributes.

```
(defstructure dac-fa owner group owner-perms group-perms other-perms)
```

```
(defalist fid->dac-fa (l) (fidp . dac-fa-p))
```

The structure `dac-pa` contains the attributes that can be associated with a process: owner, and a set of groups. The data type `pid->dac-pa` models the association of a process identifier with the process's attributes.

```
(defstructure dac-pa owner groups)
```

```
(defalist pid->dac-pa (l) (pidp . dac-pa-p))
```

The structure `dac` represents the current dac attributes assigned to files and processes.

```
(defstructure dac fdac pdac)
```

The current permissions that a process has to a file is computed by `current-perms`. This function follows the standard algorithm of checking the owner perms first, then group permissions, and finally other permissions.

```
(defun current-perms (dac pid? fid?)
  (declare (xargs :guard (and (dac-p dac) (pidp pid?) (fidp fid?))))
  (if (and (bound? pid? (dac-pdac dac)) (bound? fid? (dac-fdac dac)))
      (let ((pa (binding pid? (dac-pdac dac)))
            (fa (binding fid? (dac-fdac dac))))
        (if (equal (dac-pa-owner pa) (dac-fa-owner fa))
            (dac-fa-owner-perms fa)
            (if (member-equal (dac-fa-group fa) (dac-pa-groups pa))
                (dac-fa-group-perms fa) (dac-fa-other-perms fa))))
      nil))
```

`dacop?` defines the operations for which the function `necessary-perms` computes required permissions. `necessary-perms` also takes a `mode` argument, which makes sense only in the case where `dacop = 'open`.

```
(defun dacop? (x) (member-equal x '(open link unlink create search)))


(defun necessary-perms (dacop mode)
  (declare (xargs :guard (and (dacop? dacop) (modep mode))))
  (case dacop
    (open (case mode (rdonly '(r)) (rdwr '(r w)) (t '(w))))
    (link '(w x))
    (unlink '(w x))
    (search '(x))
    (create '(w x))
    (t nil)))
```

A pathname `pn` to a file is visible to a process if the process has permissions sufficient to search every ancestor directory along the path. `dac-visible-pn` decides of a pathname is visible to a process, given the current DAC permissions. This function is used as a guard in operations defined at the Unix level.

```
(defun dac-visible-pn (bas dac pid? startfid? pn?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (dac-p dac)
                        (pidp pid?) (fidp startfid?) (path-namep pn?)
                        (legal-pathnamep startfid? pn?
                            (fs-fcontents (bas-fspace bas))
                            (fs-ftyp (bas-fspace bas))))))
  (cond
    ((atom pn?) t)
    (t (and (subsetp-equal (necessary-perms 'search 'rdonly)
                (current-perms dac pid? startfid?))
            (dac-visible-pn bas dac pid?
                (pathname-fid startfid? (list (car pn?))
                    (fs-fcontents (bas-fspace bas)))
                (cdr pn?))))))
```

11 The Unix File System Interface

11.1 The Model

`pid->fid` is the recognizer for a mapping from process ids to file ids. This is the data type used to associate a process with its current working directory.

```
(defalist pid->fid (l) (pidp . fidp))
```

At the level of the Unix file system, state includes the basic access system `bas`, the dac state `dac`, a mapping `pcwd` that maps each process to its current working directory, and the current process `cpid`. `ufsp` recognizes a legal state at the Unix file system interface. The requirements on the argument `ufs` are

1. `ufs` satisfies `ufs-p`, which means that it is a `ufs` defstructure.
2. The basic access system is legal.
3. The dac structures are legal.
4. The process current working directory map satisfies `pid->fid`.
5. The current process is a pid.
6. Every file recorded in the DAC tables is in the file space.
7. Every process recorded in the DAC tables is in the process table.
8. Every process has a current working directory.

9. The current process is in the process table.

10. Every fid in the range of `pcwd` is a directory file in the file space. (The function `directory-fids` collects the subset of fids that are directories.)

```
(defstructure ufs bas dac pcwd cpid)
```

```
(defun unix-file-system (ufs)
  (let* ((bas (ufs-bas ufs)) (dac (ufs-dac ufs)) (pcwd (ufs-pcwd ufs))
         (cpid (ufs-cpid ufs)) (fs (bas-fspace bas)))
    (and (ufs-p ufs) (basic-access-system bas) (dac-p dac)
         (pid->fid pcwd) (pidp cpid)
         (equal (domain (dac-fdac dac)) (domain (fs-fcontents fs)))
         (equal (domain (dac-pdac dac)) (domain (bas-ptable bas)))
         (equal (domain pcwd) (domain (bas-ptable bas)))
         (member-equal cpid (domain (bas-ptable bas)))
         (subsetp-equal (range pcwd) (directory-fids (fs-ftyp fs))))))
```

We introduce relative and absolute pathnames. A *labelled pathname* is a 2-tuple consisting of a pathname type and a pathname. The function `absolute-pathname` computes an absolute pathname from a labelled pathname, using a process's current working directory as the starting point of the pathname when the label is `relative`.

```
(defun path-name-typep (x) (member-equal x '(relative absolute)))
```

```
(defstructure lpn type pathname)
```

```
(defun absolute-pathname (bas pcwd pid lpn)
  (declare (xargs :guard
                  (and (basic-access-system bas) (pidp pid) (lpn-p lpn)
                       (pid->fid pcwd))))
  (let ((fs (bas-fspace bas)))
    (if (eq (lpn-type lpn) 'relative)
        (mv (binding pid pcwd) (lpn-pathname lpn))
        (mv (fs-rootfid fs) (lpn-pathname lpn)))))
```

Following are executable models of some of the Unix file system operations. These functions model successful outcomes only. That is, preconditions on arguments are satisfied and the DAC requirements are met. These requirements are stated explicitly in the guard to each operation. A layer above, that includes return codes for various error conditions, could be constructed with no conceptual difficulty.

The predicate `create-permitted-dac` defines the conditions under which DAC permits a file creation operation. `create-file-attrs-dac` models the update to DAC permissions when a file is created. `createok-ufs` defines a successful file creation on UFS state.

```
(defun create-permitted-dac (bas dac pid? startfid? pn?)
  (declare (xargs :guard (and (basic-access-system bas) (dac-p dac))))
  (let* ((fs (bas-fspace bas))
         (fid (pathname-fid startfid? (nonlast pn?) (fs-fcontents fs))))
    (and (consp pn?)
         (dac-visible-pn bas dac pid? startfid? (nonlast pn?))
         (subsetp-equal (necessary-perms 'create 'rdwr)
             (current-perms dac pid? fid)))))
```

```
(defun create-file-attrs-dac
       (bas dac pid? startfid? pn? owner-perms? group-perms?
            other-perms?)
  (declare (xargs :guard
                  (let ((fs (bas-fspace bas)))
                    (and (basic-access-system bas) (dac-p dac)
                         (pidp pid?) (fidp startfid?) (path-namep pn?)
                         (perm-listp owner-perms?)
                         (perm-listp group-perms?)
                         (perm-listp other-perms?) (consp pn?)
                         (legal-pathnamep startfid? pn?
                             (fs-fcontents fs) (fs-ftyp fs))))))
  (let* ((fs (bas-fspace bas))
         (fid (pathname-fid startfid? pn? (fs-fcontents fs)))
         (pfid (pathname-fid startfid? (nonlast pn?) (fs-fcontents fs))))
    (update-dac dac :fdac
        (bind fid
              (make-dac-fa :owner
                  (dac-pa-owner (binding pid? (dac-pdac dac))) :group
                  (dac-fa-group (binding pfid (dac-fdac dac)))
                  :owner-perms owner-perms? :group-perms group-perms?
                  :other-perms other-perms?)
              (dac-fdac dac)))))
```

```
(defun create-ok-ufs
       (ufs lpn? ftyp? owner-perms? group-perms? other-perms?)
  (declare (xargs :guard
                  (let ((bas (ufs-bas ufs)) (dac (ufs-dac ufs))
                        (pcwd (ufs-pcwd ufs)) (cpid (ufs-cpid ufs)))
                    (and (unix-file-system ufs)
                         (mv-let (startfid pn)
                                 (absolute-pathname bas pcwd cpid lpn?)
                                 (create-permitted-dac bas dac cpid
                                     startfid pn))))))
  (let ((bas (ufs-bas ufs)) (dac (ufs-dac ufs)) (pcwd (ufs-pcwd ufs))
        (cpid (ufs-cpid ufs)))
    (mv-let (startfid pn) (absolute-pathname bas pcwd cpid lpn?)
            (let ((bas* (create-bas bas startfid pn ftyp?)))
              (update-ufs ufs :bas bas* :dac
                  (create-file-attrs-dac bas* dac cpid startfid pn
                      owner-perms? group-perms? other-perms?))))))
```

The predicate open-permitted-dac defines the conditions under which DAC permits a file open operation. open-existing-file-ok-ufs defines a successful open operation on an existing file.

```
(defun open-permitted-dac (bas dac pid? startfid? pn? mode?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (dac-p dac)
                       (pidp pid?) (fidp startfid?) (path-namep pn?)
                       (modep mode?))))
  (let* ((fs (bas-fspace bas))
         (fid (pathname-fid startfid? pn? (fs-fcontents fs))))
    (and (dac-visible-pn bas dac pid? startfid? pn?)
         (subsetp-equal (necessary-perms 'open mode?)
             (current-perms dac pid? fid)))))
```

```
(defun open-existing-file-ok-ufs (ufs lpn? mode?)
  (declare (xargs :guard
                  (let ((bas (ufs-bas ufs)) (dac (ufs-dac ufs))
                        (pcwd (ufs-pcwd ufs)) (cpid (ufs-cpid ufs)))
                    (and (unix-file-system ufs)
                         (mv-let (startfid pn)
                                 (absolute-pathname bas pcwd cpid lpn?)
                                 (open-permitted-dac bas dac cpid
                                     startfid pn mode?))))))
  (let ((bas (ufs-bas ufs)) (pcwd (ufs-pcwd ufs))
        (cpid (ufs-cpid ufs)))
    (mv-let (startfid pn) (absolute-pathname bas pcwd cpid lpn?)
            (mv-let (fd bas*) (open-bas bas cpid startfid pn mode?)
                    (mv fd (update-ufs ufs :bas bas*))))))
```

The predicate `link-permitted-dac` defines the conditions under which DAC permits a file link operation. `link-ok-ufs` defines a successful link operation.

```
(defun link-permitted-dac (bas dac pid? startfid? pn? xstartfid? xpn?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (dac-p dac)
                       (pidp pid?) (fidp startfid?) (path-namep pn?)
                       (fidp xstartfid?) (path-namep xpn?))))
  (let* ((fs (bas-fspace bas))
         (fid (pathname-fid startfid? (nonlast pn?) (fs-fcontents fs))))
    (and (consp pn?)
         (dac-visible-pn bas dac pid? startfid? (nonlast pn?))
         (dac-visible-pn bas dac pid? xstartfid? xpn?)
         (subsetp-equal (necessary-perms 'link 'rdwr)
             (current-perms dac pid? fid)))))


(defun link-ok-ufs (ufs lpn? xlpn?)
  (declare (xargs :guard
                  (let ((bas (ufs-bas ufs)) (dac (ufs-dac ufs))
                        (pcwd (ufs-pcwd ufs)) (cpid (ufs-cpid ufs)))
                    (and (unix-file-system ufs)
                         (mv-let (startfid pn)
                                 (absolute-pathname bas pcwd cpid lpn?)
                                 (mv-let (xstartfid xpn)
                                         (absolute-pathname bas pcwd
                                          cpid xlpn?)
                                         (link-permitted-dac bas dac
                                          cpid startfid pn xstartfid
                                          xpn)))))))
  (let ((bas (ufs-bas ufs)) (pcwd (ufs-pcwd ufs))
        (cpid (ufs-cpid ufs)))
    (mv-let (startfid pn) (absolute-pathname bas pcwd cpid lpn?)
            (mv-let (xstartfid xpn)
                    (absolute-pathname bas pcwd cpid xlpn?)
                    (update-ufs ufs :bas
                                (link-bas bas startfid pn xstartfid
                                    xpn))))))
```

The predicate `unlink-permitted-dac` defines the conditions under which DAC permits a file unlink operation. `unlink-ok-ufs` defines a successful unlink operation.

```
(defun unlink-permitted-dac (bas dac pid? startfid? pn?)
  (declare (xargs :guard
                  (and (basic-access-system bas) (dac-p dac)
                       (pidp pid?) (fidp startfid?) (path-namep pn?))))
  (let* ((fs (bas-fspace bas))
         (fid (pathname-fid startfid? (nonlast pn?) (fs-fcontents fs))))
    (and (consp pn?)
         (dac-visible-pn bas dac pid? startfid? (nonlast pn?))
         (subsetp-equal (necessary-perms 'unlink 'rdwr)
             (current-perms dac pid? fid)))))



(defun unlink-ok-ufs (ufs lpn?)
  (declare (xargs :guard
                  (let ((bas (ufs-bas ufs)) (dac (ufs-dac ufs))
                        (pcwd (ufs-pcwd ufs)) (cpid (ufs-cpid ufs)))
                    (and (unix-file-system ufs)
                         (mv-let (startfid pn)
                                 (absolute-pathname bas pcwd cpid lpn?)
                                 (unlink-permitted-dac bas dac cpid
                                     startfid pn))))))
  (let ((bas (ufs-bas ufs)) (pcwd (ufs-pcwd ufs))
        (cpid (ufs-cpid ufs)))
    (mv-let (startfid pn) (absolute-pathname bas pcwd cpid lpn?)
            (update-ufs ufs :bas (unlink-bas bas startfid pn)))))
```

No explicit DAC permissions are required for read, write and close operations. DAC checks are made at the time a file is opened.

```
(defun read-ok-ufs (ufs fd? length?)
  (declare (xargs :guard
                  (and (unix-file-system ufs) (fdp fd?)
                       (naturalp length?))))
  (let ((bas (ufs-bas ufs)) (cpid (ufs-cpid ufs)))
    (mv-let (data bas*) (read-bas bas cpid fd? length?)
            (mv data (update-ufs ufs :bas bas*)))))
```

```
(defun write-ok-ufs (ufs fd? data?)
  (declare (xargs :guard
                  (let* ((bas (ufs-bas ufs)) (cpid (ufs-cpid ufs))
                         (ft (bas-ftable bas))
                         (oid (fd-to-oid (bas-ptable bas) cpid fd?)))
                    (and (unix-file-system ufs) (fdp fd?) (filep data?)
                         (or (opened-for-write ft oid)
                             (opened-for-append ft oid))))))
  (let* ((bas (ufs-bas ufs)) (cpid (ufs-cpid ufs))
         (ft (bas-ftable bas))
         (oid (fd-to-oid (bas-ptable bas) cpid fd?)))
    (if (opened-for-write ft oid)
        (update-ufs ufs :bas (write-bas bas cpid fd? data?))
        (update-ufs ufs :bas (append-bas bas cpid fd? data?)))))
```

```
(defun close-ok-ufs (ufs fd?)
  (declare (xargs :guard (and (unix-file-system ufs) (fdp fd?))))
  (let ((bas (ufs-bas ufs)) (cpid (ufs-cpid ufs)))
    (update-ufs ufs :bas (close-bas bas cpid fd?))))
```

11.2 Evaluation of the Model

Here are some calls that illustrate the operation of the Unix File System interface. We let `bas10`, constructed in Section 9.2 be our initial BAS state. Recall that this state includes the directories `/usr/jones` and `/usr/smith`, and the regular file `/usr/jones/mail`. No files are open. There are two processes, 100 and 200.

```
(equal (@ bas10)
       '(bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2) (3)
                  (4 mail 6) (5) (6 a b c d e f g h i j k l m n o p))
                 ((0 . dir) (1 . dir) (2 . dir) (3 . dir) (4 . dir)
                  (5 . dir) (6 . reg))
                 0)
             (ft nil nil nil) ((100) (200))))
```

`dac0` is a DAC state constructed for these files and processes. Process 100 is assigned to Jones, and 200 to Smith. Jones is the owner of `/usr/jones`, and Smith is the owner of `/usr/smith`. `r` and `x` permissions to these directories are granted to other users. No one other than Jones has access to `/usr/jones/mail`.

```
(equal (@ dac0)
       '(dac ((0 dac-fa root nil (r w x) nil (r x))
              (1 dac-fa root nil (r w x) nil (r w x))
              (2 dac-fa root nil (r w x) nil (r w x))
              (3 dac-fa root nil (r w x) nil (r w x))
              (4 dac-fa jones nil (r w x) nil (r x))
              (5 dac-fa smith nil (r w x) nil (r x))
              (6 dac-fa jones nil (r w x) nil nil))
             ((100 dac-pa jones nil) (200 dac-pa smith nil))))
```

The initial working directory table assigns `/usr/jones` to process 100, and `/usr/smith` to process 200.

```
(assign pcwd0 '((100 . 4) (200 . 5)))
```

The initial state `ufs0` for our example includes `bas0`, `dac0`, `pcwd0`, and 100 is the current process. This state satisfies the legal state predicate for the Unix file system interface.

```
(assign ufs0
        (make-ufs :bas (@ bas10) :dac (@ dac0) :pcwd (@ pcwd0) :cpid
            100))

(equal (unix-file-system (@ ufs0)) t)
```

All pathnames in this state are visible to both Jones and Smith. Here are some evaluations of the visible pathname function.

```
(equal (dac-visible-pn (@ bas10) (@ dac0) 100 0 '(usr jones mail)) t)

(equal (dac-visible-pn (@ bas10) (@ dac0) 200 0 '(usr jones mail)) t)
```

We create a subdirectory `/usr/jones/secret`, and then `/usr/jones/secret/plans`. The latter pathname is not visible to Smith, which is illustrated below.

```
(assign ufs1
        (create-ok-ufs (@ ufs0)
            (make-lpn :type 'relative :pathname '(secret)) 'dir
            '(r w x) nil nil))
```

```
(equal (@ ufs1)
       '(ufs (bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2)
                       (3) (4 mail 6 secret 7) (5)
                       (6 a b c d e f g h i j k l m n o p) (7))
                      ((0 . dir) (1 . dir) (2 . dir) (3 . dir)
                       (4 . dir) (5 . dir) (6 . reg) (7 . dir))
                      0)
                  (ft nil nil nil) ((100) (200)))
             (dac ((0 dac-fa root nil (r w x) nil (r x))
                   (1 dac-fa root nil (r w x) nil (r w x))
                   (2 dac-fa root nil (r w x) nil (r w x))
                   (3 dac-fa root nil (r w x) nil (r w x))
                   (4 dac-fa jones nil (r w x) nil (r x))
                   (5 dac-fa smith nil (r w x) nil (r x))
                   (6 dac-fa jones nil (r w x) nil nil)
                   (7 dac-fa jones nil (r w x) nil nil))
                  ((100 dac-pa jones nil) (200 dac-pa smith nil)))
             ((100 . 4) (200 . 5)) 100))

(assign ufs2
        (create-ok-ufs (@ ufs1)
            (make-lpn :type 'relative :pathname '(secret plans)) 'reg
            '(r w x) nil nil))

(equal (@ ufs2)
       '(ufs (bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2)
                       (3) (4 mail 6 secret 7) (5)
                       (6 a b c d e f g h i j k l m n o p) (7 plans 8)
                       (8))
                      ((0 . dir) (1 . dir) (2 . dir) (3 . dir)
                       (4 . dir) (5 . dir) (6 . reg) (7 . dir)
                       (8 . reg))
                      0)
                  (ft nil nil nil) ((100) (200)))
             (dac ((0 dac-fa root nil (r w x) nil (r x))
                   (1 dac-fa root nil (r w x) nil (r w x))
                   (2 dac-fa root nil (r w x) nil (r w x))
                   (3 dac-fa root nil (r w x) nil (r w x))
                   (4 dac-fa jones nil (r w x) nil (r x))
                   (5 dac-fa smith nil (r w x) nil (r x))
                   (6 dac-fa jones nil (r w x) nil nil)
                   (7 dac-fa jones nil (r w x) nil nil)
                   (8 dac-fa jones nil (r w x) nil nil))
                  ((100 dac-pa jones nil) (200 dac-pa smith nil)))
             ((100 . 4) (200 . 5)) 100))
```

Now we see that `/usr/jones/secret/plans` is not a pathname visible to Smith.

```
(equal (not (dac-visible-pn (ufs-bas (@ ufs2)) (ufs-dac (@ ufs2)) 200 0
                '(usr jones secret plans)))
       t)
```

We illustrate one more call at the ufs level, unlinking `/usr/jones/mail`. This causes file 6 to be garbage collected from the file space, since there are no references to it. However, we have not implemented garbage collection for the DAC state, and therefore the file attributes for file 6 remain.

```
(assign ufs3
        (unlink-ok-ufs (@ ufs2)
            (make-lpn :type 'relative :pathname '(mail))))


(equal (@ ufs3)
       '(ufs (bas (fs ((0 usr 1 tmp 2 dev 3) (1 jones 4 smith 5) (2)
                        (3) (4 secret 7) (5) (7 plans 8) (8))
                       ((0 . dir) (1 . dir) (2 . dir) (3 . dir)
                        (4 . dir) (5 . dir) (7 . dir) (8 . reg))
                       0)
                   (ft nil nil nil) ((100) (200)))
             (dac ((0 dac-fa root nil (r w x) nil (r x))
                    (1 dac-fa root nil (r w x) nil (r w x))
                    (2 dac-fa root nil (r w x) nil (r w x))
                    (3 dac-fa root nil (r w x) nil (r w x))
                    (4 dac-fa jones nil (r w x) nil (r x))
                    (5 dac-fa smith nil (r w x) nil (r x))
                    (6 dac-fa jones nil (r w x) nil nil)
                    (7 dac-fa jones nil (r w x) nil nil)
                    (8 dac-fa jones nil (r w x) nil nil))
                   ((100 dac-pa jones nil) (200 dac-pa smith nil)))
             ((100 . 4) (200 . 5)) 100))
```

References

[BCT95] William R. Bevier, Richard M. Cohen, and Jeff Turner.
A specification for the synergy file system.
Technical report, Computational Logic, Inc., September 1995.

[KM94] Matt Kaufmann and J Strother Moore.
Design goals of acl2.
Technical Report 101, Computational Logic, Inc., August 1994.

[Spi89] J.M. Spivey.
The Z Notation: A Reference Manual.
Prentice Hall, 1989.

[Ste84] Guy L. Steele Jr.
Common LISP: The Language.
Digital Press, 1984.

[Ste90] Guy L. Steele Jr.
Common LISP: The Language, Second Edition.
Digital Press, 1990.

Index