Verified File-System v1.0

Simão Cunha, Luís Rodrigues, Augusto Silva, Rui Gonçalves, Samuel Silva
Formal Methods II (Mathematics and Computer Science degree)
Department of Informatics - University of Minho

Braga, September 27, 2007

Contents

1 Introduction

The verification grand challenge proposed by Prof. Tony Hoare sets the stage for the program verification community to embark upon a collaborative effort to build verifiable programs. In recent meetings it has been suggested that the community should concentrate on the verification of the Linux kernel, but this would be an impossible task to do in a couple of years due to this kernel's size. So, Rajeev Joshi and Gerald Holzmann [3] decided to propose a "mini challenge": build a verifiable file system.

There are several reasons why a file system is more attractive as first target for verification. Firstly, most modern file systems have a well-defined interface conforming to the POSIX standard. Thus, writing a formal specification for a POSIX-compliant file system would require far less effort than writing a kernel specification. Secondly, the algorithms and data structures used in file system design are well understood and a verifiable file system implementation could be written from scratch. Alternatively, researchers could choose an existing file system and attempt to verify it. Thirdly, although file systems are only a small portion of an operating system, they are complex enough for ensuring reliability in the presence of concurrent access, unexpected power or hardware failure etc to be nontrivial. Finally, since almost all data on modern computers are now managed by file systems, their correctness is of great importance, both from the standpoint of reliability as well as security. Development of a verified file system would therefore be of great value.

Most modern file systems are written to comply with the POSIX standard for file systems. This standard specifies a set of function signatures, along with a behavioral description of each function. However, these behavioral descriptions are given as informal English prose, and are therefore too ambiguous and incomplete to be useful in a verification effort. So we have writen a formal specification of some POSIX standard functions (eg. mkdir, open, read, write) using the modeling language VDM++ [9], although some features of the POSIX standard are not completely modeled.

2 Analysis and Development

We divided this assignment in two steps. The first is the definition of data types and the second the functions that manipulate them. We will first develop the functions in an algebric way. After that, we'll use these functions to build an object that will behave like a file system. To help us on our work we used the requirements catalog available at the OLVER project [5]. From all the requirements some were discharged due to the difficulty or extension of specifying them in VDM. All requirements related with time, concurrency, special files (pipes, sockets, terminal devices, etc.) and signals have been deliberately "forgotten" in the current revion of the formal model.

2.1 Data Types

The first step for the problem resolution is to define the necessary data types.

2.1.1 Tar

There are several ways of implementing a file system, for this assignment we used a non-recursive version known as 'Tar' or tape archive. It's a 'map' that has 'paths' as keys and a 'file' associated with each path. An invariant is added to enforce that all files and directories sub-paths are present in 'Tar' and sub-paths must be of directory type. Also a root is needed for Tar.

```
class Tar
 Id = char*
inv id ≜
    id ≠ [] ∧
    elems id ⊆ ({'-','_','.','(',')'} ∪
     {'0','1','2','3','4','5','6','7','8','9'} ∪
     {'a','b','c','d','e','f','g','h','i','j','k','l','m'} ∪
     {'n','o','p','q','r','s','t','u','v','w','x','y','z'} ∪
     {'A','B','C','D','E','F','G','H','I','J','K','L','M'} ∪
     {'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'});

Path :: path : Tar`Id*;
 Tar = Tar`Path →m Tar`File
inv tar ≜
    ∀ p ∈ dom tar ·
        (subpaths (p) ⊆ (dom tar) ∧ ∀ x ∈ (subpaths (p)) · tar (x).attrib.type = D) ∧
        (mk-Tar`Path ([]) ∈ dom tar ∧ tar (mk-Tar`Path ([])).attrib.type = D);
```

2.1.2 File

A 'file' type is identified by it's attribute (attrib.type) that may be the following values: a regular file (<r>) , a directory (<d>), a link (<l>) , a socket (<s>), a character device file special (<c>) or a block device special file (<b>). To simplify the problem we have chosen that our function would only work with regular files, directories or links. User id and group id of the owner as well as permission bits that indicate what permissions the owner, the group or others have are also part of the file attributes. If the file is a directory the links fields indicates the number of files or directories that directory contains. The size field indicates the total size of the file. The dates field is divided into three other fields, representing the file last access (st_atime), last data modification (st_mtime) and last i-node modification (st_ctime). Each of these date fields contains a date and a flag that shows if the date needs to be updated. The date can also be *nil* to cover the case where it was still not update, for example, when a file or directory is created, it takes a first update to fill the dates.

The file contents depends on which type the file is, if the file is a directory then the contents is nil, in case of being a regular file, the contents is a 'map' from address numbers to 'Byte', if the file is a link, the contents is the path that the link points and if type is block or character device, the file contents indicates to what device is the file associated. An invariant was used in the type 'File' to enforce that it is well formed concerning the file contents. The invariant is commented because although it was working in VDM-SL, we had problems using it in VDM++ (maybe it's a VDM-tools bug).

```
FileType = R | D | L | S | P | C | B;

FilePermissions :: user : Tar`Permissions
                   group : Tar`Permissions
                   other : Tar`Permissions;

Permissions :: read : 𝔹
               write : 𝔹
               exec : 𝔹;
UserId = ℕ;
GroupId = ℕ;

Attributes :: type : Tar`FileType
              perm : Tar`FilePermissions
              links : ℕ
              user : Tar`UserId
              group : Tar`GroupId
              size : ℕ
              dates : Dates;

Dates :: st-atime : Date
         st-ctime : Date
         st-mtime : Date;

Date :: date : [token]
        update : 𝔹;

FileContents :: content : Tar`Memory;
Device = ℕ;

File :: attrib : Tar`Attributes
        contents : [Tar`FileContents | Tar`Path | Tar`Device];
```

2.1.3 System

To implement the functions we needed far more than only the 'Tar' type. It was necessary a type to model the operating system in which the file system is embedded. That type is the 'System' and beside a filesystem 'Tar' it also includes a table of the processes currently running on the system, a file descriptor table, the actual process running on the system, the system constants, the errno variable, the unused space of the file system, the system main memory and a table of opened devices.

The process table contains various information regarding the processes that are running in the system, like ownership (user id and group id), the file permission mask associated with process and it's used file descriptors.

The FileDescriptor associates a file descriptor (nat) with an OpenFileDescriptor (OFD), the 'OFD' contains a path to a file, the status of the OFD (if 'append' is set), various flags used in the implemented functions, the seek offset, the maximum offset supported by the system and the mode that the file was opened (read, write, read & write).

The constants table contains the values of constants that are needed for the system to work.

The errno field is the field that holds the error message when a function fails.

The unused space field indicates how much free space the file system has.

The environment contains various flags that represent various situations that can happen in the system, like if the system is unable to allocate resources or a socket. A field indicating what mode is the file system mounted.

$$
\begin{array}{l}
Constants = \mathsf{char}^{+} \xrightarrow{m} \mathbb{Z}; \\
FileDescriptors = \mathbb{N} \xrightarrow{m} Tar`OpenFileDescriptor; \\
\\
OpenFileDescriptor :: file : Tar`Path \\
\qquad\qquad\qquad\qquad\quad status : Flags\text{-}\mathsf{set} \\
\qquad\qquad\qquad\qquad\quad offset : \mathbb{N} \\
\qquad\qquad\qquad\qquad\quad maxoffset : \mathbb{N} \\
\qquad\qquad\qquad\qquad\quad flags : Tar`FDFlags\text{-}\mathsf{set} \\
\qquad\qquad\qquad\qquad\quad mode : \text{RD} \mid \text{WR} \mid \text{RDWR}; \\
FDFlags = \text{FD\_CLOEXEC} \mid \text{ISIG} \mid \text{ICANON} \mid \text{IXON} \mid \text{ICRNL} \mid \text{IGNCR} \mid \text{INLCR}; \\
\\
ProcInfo :: userid : Tar`UserId \\
\qquad\qquad groupid : Tar`GroupId \\
\qquad\qquad umask : Tar`FilePermissions \\
\qquad\qquad usedfds : \mathbb{N}\text{-}\mathsf{set}; \\
\\
Environment :: signal : \mathbb{B} \\
\qquad\qquad\qquad\quad unable2socket : \mathbb{B} \\
\qquad\qquad\qquad\quad unable2resorces : \mathbb{B} \\
\qquad\qquad\qquad\quad hangup : \mathbb{B} \\
\qquad\qquad\qquad\quad fsmode : \text{RD} \mid \text{WR} \mid \text{RDWR} \\
\qquad\qquad\qquad\quad physicalerror : \mathbb{B}; \\
Byte = \mathsf{char} \mid Tar`Trash; \\
Trash = \mathsf{token}; \\
Memory = \mathbb{N} \xrightarrow{m} Tar`Byte; \\
\\
System :: tarFileSystem : Tar`Tar \\
\qquad\quad process : \mathbb{N} \xrightarrow{m} ProcInfo \\
\qquad\quad fds : Tar`FileDescriptors \\
\qquad\quad actualproc : \mathbb{N} \\
\qquad\quad constants : Constants \\
\qquad\quad errno : \mathsf{char}^{*} \\
\qquad\quad unusedspace : \mathbb{N} \\
\qquad\quad environment : Tar`Environment \\
\qquad\quad memory : Tar`Memory \\
\qquad\quad devices : Tar`Device \xrightarrow{m} \mathsf{token} \\
\mathsf{inv}\ s \triangleq \\
\qquad s.actualproc \in \mathsf{dom}\ s.process;
\end{array}
$$

2.1.4 Mode_t and Flags

Mode_t These are flags to be used as arguments of *mkdir* and *open* functions and what they mean is respectively:

- read, write, execute/search by owner
- read permission, owner
- write permission, owner
- execute/search permission, owner
- read, write, execute/search by group
- read permission, group
- write permission, group
- execute/search permission, group
- read, write, execute/search by others
- read permission, others
- write permission, others

- execute/search permission, others
- set-user-ID on execution
- set-group-ID on execution
- on directories, restricted deletion flag

Mode-t = S_IRWXU | S_IRUSR | S_IWUSR | S_IXUSR | S_IRWXG | S_IRGRP | S_IWGRP | S_IXGRP | S_IRWXO | S_IROTH | S_IWOTH | S_IXOTH | S_ISUID | S_ISGID | S_ISVTX;

Flags These flags are to be used as argument to *open* and what they mean is respectively:

- Open for reading only
- Open for writing only
- Open for reading and writing
- Used with communications devices
- Used with communications devices
- Open at end of file for writing
- If set this results in write() calls not returning until data has actually been written to dics.
- Create the file if it doesn't already exist
- If set and O_CREAT set will cause open() to fail if the file already exists
- Truncate file size to zero if it already exists
- If the named file is a terminal device, don't make it the controlling terminal for the process.

Flags = O_RDONLY | O_WRONLY | O_RDWR | O_NDELAY | O_NONBLOCK | O_APPEND | O_SYNC | O_CREAT | O_EXCL | O_TRUNC | O_NOCTTY

Other data types that were not mentioned are secondary and pretty self-explaining so we won't get into full details here.

2.2 Functions

2.2.1 Mkdir

mkdir NAME: mkdir – make a directory file

```
int mkdir(const char *path, mode_t mode);
```

[mkdir.01] The mkdir() function shall create a new directory with name path. [mkdir.02] The file permission bits of the new directory shall be initialized from mode. [mkdir.03] These file permission bits of the mode argument shall be modified by the process' file creation mask [mkdir.04] The directory's user ID shall be set to the process' effective user ID [mkdir.05] The directory's group ID shall be set to the group ID of the parent directory or to the effective group ID of the process. Implementations shall provide a way to initialize the directory's group ID to the group ID of the

parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the directory's group ID to the effective group ID of the calling process.

[mkdir.08] Upon successful completion, mkdir() shall mark fordd update the st_atime, st_ctime, and st_mtime fields of the directory. [mkdir.09] Also, the st_ctime and st_mtime fields of the directory that contains the new entry shall be marked for update.

[mkdir.10] Upon successful completion, mkdir() shall return 0. [mkdir.11] Otherwise, -1 shall be returned, no directory shall be created, and errno shall be set to indicate the error.

```
mkdir : Path × Mode-t-set × System → (ℤ × System)
mkdir (unresolvedpath, mode, sys) △
  let proc = sys.process (sys.actualproc),
      tar = sys.tarFileSystem,
      fileperm = mode2perm (mode),
      err = mkdirError (unresolvedpath, sys) in
  if err ≠ []
  then mk- (− 1, μ (sys, errno ↦ err))
  else let path = resolveLink (unresolvedpath, tar),
           parent = tar (blast (path)),
           date = mk-Tar`Dates
                    (
                     mk-Tar`Date (nil , true),
                     mk-Tar`Date (nil , true),
                     mk-Tar`Date (nil , true)),
           newdir = mk-Tar`File
                      (
                       mk-Tar`Attributes (D,
                                          applyMask (fileperm, proc.umask),
                                          0, proc.userid,
                                          proc.groupid,
                                          1, date),
                       nil ),
           newparentdates = mk-Tar`Dates
                              (
                               parent.attrib.dates.st-atime,
                               mk-Tar`Date (parent.attrib.dates.st-ctime.date, true),
                               mk-Tar`Date (parent.attrib.dates.st-mtime.date, true)),
           pattrib = parent.attrib,
           newparentattrib = mk-Tar`Attributes
                               (
                                pattrib.type,
                                pattrib.perm,
                                pattrib.links + 1,
                                pattrib.user,
                                pattrib.group,
                                pattrib.size + 1,
                                newparentdates),
           newparent = μ (parent, attrib ↦ newparentattrib) in
       mk- (0, μ (sys,

                  tarFileSystem ↦ (sys.tarFileSystem †
                   {blast (path) ↦ newparent, path ↦ newdir})));
```

mkdirError This function perform error detection.

[mkdir.12] The mkdir() function shall fail if:

ENOENT *[mkdir.12.06] A component of the path prefix specified by path does not name an existing directory or path is an empty string.*

ELOOP *[mkdir.12.03] A loop exists in symbolic links encountered during resolution of the path argument.*

ENOTDIR *[mkdir.12.08] A component of the path prefix is not a directory.*

EACCES *[mkdir.12.01] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.*

EEXIST *[mkdir.12.02] The named file exists.*

EMLINK *[mkdir.12.04] The link count of the parent directory would exceed LINK_MAX.*

ENAMETOOLONG *[mkdir.12.05] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.*

ENOSPC *[mkdir.12.07] The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.*

ENOTDIR *[mkdir.12.08] A component of the path prefix is not a directory.*

EROFS *[mkdir.12.09] The parent directory resides on a read-only file system.*

[mkdir.13] The mkdir() function may fail if:

ELOOP *[mkdir.13.01] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.*

ENAMETOOLONG *[mkdir.13.02] As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded PATH_MAX.*

```
mkdirError : Path × System → char*
mkdirError (unresolvedpath, sys) ≙
  let proc = sys.process (sys.actualproc),
      tar = sys.tarFileSystem,
      c = sys.constants in
  if (pathIsEmpty (unresolvedpath))
  then "[ENOENT]"
  elseif (isLoop (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (symLoop (c ("{SYMLOOP_MAX}")) (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (¬ isValidPath (blast (unresolvedpath), tar))
  then "[ENOTDIR]"
  else let path = resolveLink (unresolvedpath, tar) in
       if (cannotAccess (path, tar, proc.userid, proc.groupid))
       then "[EACCESS]"
       elseif (fileExists (path, tar))
       then "[EEXIST]"
       elseif (isLinkFull (path, tar, c ("{LINK_MAX}")))
       then "[EMLINK]"
       elseif (isNameTooLong (path, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (isNameTooLong (unresolvedpath, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (notEnoughSpaceInFs (sys))
       then "[ENOSPC]"
       elseif (prefIsNotDir (path, tar))
       then "[ENOTDIR]"
       elseif (isRdOnly (blast (path), tar, proc.userid, proc.groupid))
       then "[EROFS]"
       else [];
```

[mkdir.07] If path names a symbolic link, mkdir() shall fail and set errno to [EEXIST].

```
isSymbolicLink : Tar‘Path × Tar‘Tar → 𝔹
isSymbolicLink (p, tar) ≙
  if (p ∈ dom tar)
  then tar (p).attrib.type = L
  else false;
```

[mkdir.12.01] [EACCES] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.

```
cannotAccess : Tar`Path × Tar`Tar × Tar`UserId × Tar`GroupId → 𝔹
cannotAccess (path, tar, userid, groupid) △
   (∃ p ∈ subpaths (path) ·
         ¬ checkExecPerm (p, tar, userid, groupid)) ∨
   (¬ checkWritePerm (blast (path), tar, userid, groupid));
```

[mkdir.12.02] [EEXIST] The named file exists.

```
fileExists : Tar`Path × Tar`Tar → 𝔹
fileExists (p, tar) △
   p ∈ dom tar;
```

[mkdir.12.03] [ELOOP] A loop exists in symbolic links encountered during resolution of the path argument.

```
isLoop : Tar`Path × Tar`Tar → 𝔹
isLoop (path, tar) △
   if path.path = []
   then false
   else isLoopAux (mk-Tar`Path ([hd path.path]),
                   mk-Tar`Path (tl path.path), tar, {});

isLoopAux : Tar`Path × Tar`Path × Tar`Tar × Tar`Path-set → 𝔹
isLoopAux (resolve, rest, tar, accum) △
   if resolve ∉ dom tar
   then false
   else if (tar (resolve).attrib.type = L)
        then let newpath = tar (resolve).contents in
             if (newpath = resolve ∨ newpath ∈ accum)
             then true
             else cases newpath :
                     mk-Tar`Path ([]) →
                           isLoopAux (mk-Tar`Path ([]),
                                      mk-Tar`Path (rest.path), tar, {}),
                     others → isLoopAux (mk-Tar`Path ([hd newpath.path]),
                                         mk-Tar`Path (newpath.path ⌢ rest.path),
                                         tar, {resolve} ∪ accum)
                  end
        else if (tar (resolve).attrib.type = D)
             then cases rest :
                     mk-Tar`Path ([]) → false,
                     others → isLoopAux
                                 (
                                  mk-Tar`Path (resolve.path ⌢ [hd rest.path]),
                                  mk-Tar`Path (tl rest.path), tar, {})
                  end
             else false;
```

[mkdir.12.04] [EMLINK] The link count of the parent directory would exceed LINK_MAX.

```
isLinkFull : Tar`Path × Tar`Tar × ℕ → 𝔹
isLinkFull (path, tar, link-max) △
   tar (blast (path)).attrib.links > link-max
pre  blast (path) ∈ dom tar
   ;
```

[mkdir.12.05] [ENAMETOOLONG] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX. [mkdir.13.02] [ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded PATH_MAX.

```
isNameTooLong : Tar`Path × ℕ × ℕ → 𝔹
isNameTooLong (p, path-max, name-max) △
   len p.path ≥ path-max ∨ len last (p) > name-max;
```

[mkdir.12.06] [ENOENT] A component of the path prefix specified by path does not name an existing directory or path is an empty string.

```
pathIsEmpty : Tar‘Path → 𝔹
pathIsEmpty (path) △
    path.path = [];
```

[mkdir.12.07] [ENOSPC] The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.

```
notEnoughSpaceInFs : System → 𝔹
notEnoughSpaceInFs (sys) △
    sys.unusedspace = 0;
```

[mkdir.12.08] [ENOTDIR] A component of the path prefix is not a directory.

```
prefIsNotDir : Tar‘Path × Tar‘Tar → 𝔹
prefIsNotDir (path, tar) △
    ∃ p ∈ subpaths (path) · ¬ (tar (p).attrib.type = D);
```

[mkdir.12.09] [EROFS] The parent directory resides on a read-only file system.

```
isRdOnly : Tar‘Path × Tar‘Tar × ℕ × ℕ → 𝔹
isRdOnly (path, tar, userid, groupid) △
    checkRDOnly (path, tar, userid, groupid)
pre  path ∈ dom tar
    ;
```

[mkdir.13.01] [ELOOP] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.

```
symLoop : ℕ → Tar‘Path × Tar‘Tar → 𝔹
symLoop (symloop-max)(path, tar) △
    if path.path = []
    then false
    else verifySymLoop (symloop-max,
                        mk-Tar‘Path ([hd path.path]),
                        tar, 0,
                        mk-Tar‘Path (tl path.path));

verifySymLoop : ℕ × Tar‘Path × Tar‘Tar × ℕ × Tar‘Path → 𝔹
verifySymLoop (max, path, tar, count, rest) △
    if count = max
    then true
    else if path ∉ dom tar
         then false
         else if tar (path).attrib.type = L
              then let newpath = tar (path).contents in
                   cases newpath :
                     mk-Tar‘Path ([]) →
                           verifySymLoop (max, mk-Tar‘Path ([]),
                                          tar, count + 1,
                                          mk-Tar‘Path (rest.path)),
                     others → verifySymLoop (max,
                                             mk-Tar‘Path ([hd newpath.path]),
                                             tar, count + 1,
                                             mk-Tar‘Path (tl newpath.path ↷ rest.path))
                   end
              else cases rest :
                     mk-Tar‘Path ([]) → false,
                     others → verifySymLoop (max,
                                             mk-Tar‘Path (path.path ↷ [hd rest.path]),
                                             tar, count,
                                             mk-Tar‘Path (tl rest.path))
                   end;
```

[mkdir.12.08] [ENOTDIR] A component of the path prefix is not a directory.

```
isValidPath : Tar`Path × Tar`Tar → 𝔹
isValidPath (path, tar) △
   if path.path = []
   then true
   else isValidPathAux
              (
               mk-Tar`Path ([hd path.path]),
               mk-Tar`Path (tl path.path), tar);

isValidPathAux : Tar`Path × Tar`Path × Tar`Tar → 𝔹
isValidPathAux (path, rest, tar) △
   if path ∉ dom tar
   then false
   else cases rest :
           mk-Tar`Path ([]) → true,
           others → let file = tar (path) in
                    if file.attrib.type = L
                    then isValidPathAux
                                (
                                 mk-Tar`Path ([hd file.contents.path]),
                                 mk-Tar`Path (tl file.contents.path ↷ rest.path), tar)
                    else isValidPathAux
                                (
                                 mk-Tar`Path (path.path ↷ [hd rest.path]),
                                 mk-Tar`Path (tl rest.path), tar)
        end
pre  path.path ≠ []
   ;
```

This function perform a resolve link.

```
resolveLink : Tar`Path × Tar`Tar → Tar`Path
resolveLink (p, tar) △
  if p.path = []
  then p
  else resolveLinkAux
            (
             mk-Tar`Path ([hd p.path]),
             mk-Tar`Path (tl p.path), tar)
pre  if p.path ≠ []
     then isValidPath (blast (p), tar)
     else true
  ;

resolveLinkAux : Tar`Path × Tar`Path × Tar`Tar → Tar`Path
resolveLinkAux (path, rest, tar) △
  if path ∉ dom tar
  then path
  else cases rest :
         mk-Tar`Path ([]) →
              if (tar (path).attrib.type = L)
              then let newpath = tar (path).contents in
                   cases newpath :
                      mk-Tar`Path ([]) → mk-Tar`Path ([]),
                      others → resolveLinkAux
                                    (
                                     mk-Tar`Path ([hd newpath.path]),
                                     mk-Tar`Path (tl newpath.path), tar)
                   end
              else path,
         others → if tar (path).attrib.type = L
                  then let newpath = tar (path).contents in
                       cases newpath :
                          mk-Tar`Path ([]) →
                                resolveLinkAux (mk-Tar`Path ([]), rest, tar),
                          others → resolveLinkAux
                                        (
                                         mk-Tar`Path ([hd newpath.path]),
                                         mk-Tar`Path (tl newpath.path ⌢ rest.path), tar)
                       end
                  else resolveLinkAux
                            (
                             mk-Tar`Path (path.path ⌢ [hd rest.path]),
                             mk-Tar`Path (tl rest.path), tar)
       end;
```

2.2.2 Rmdir

rmdir NAME: rmdir – remove a directory

```
int rmdir(const char *path);
```

[rmdir.01] The rmdir() function shall remove a directory whose name is given by path. [rmdir.06] Upon successful completion, the rmdir() function shall mark for update the st_ctime and st_mtime fields of the parent directory. [rmdir.07] Upon successful completion, the function rmdir() shall return 0. [rmdir.08] Otherwise, -1 shall be returned, and errno set to indicate the error. If -1 is returned, the named directory shall not be changed.

```
rmdir : Path × System → (ℤ × System)
rmdir (unresolvedpath, sys) ≙
  let tar = sys.tarFileSystem,
      err = rmdirError (unresolvedpath, sys) in
  if err ≠ []
  then mk- (− 1, μ (sys, errno ↦ err))
  else let path = resolveLink (unresolvedpath, tar),
           parent = tar (blast (path)),
           newparentdates = mk-Tar`Dates
                                (
                                 parent.attrib.dates.st-atime,
                                 mk-Tar`Date (parent.attrib.dates.st-ctime.date, true),
                                 mk-Tar`Date (parent.attrib.dates.st-mtime.date, true)),
           pattrib = parent.attrib,
           newparentattrib = mk-Tar`Attributes
                                (
                                 pattrib.type,
                                 pattrib.perm,
                                 pattrib.links − 1,
                                 pattrib.user,
                                 pattrib.group,
                                 pattrib.size − 1,
                                 newparentdates),
           newparent = μ (parent, attrib ↦ newparentattrib) in
       mk- (0, μ (sys,

                  tarFileSystem ↦ {path} ⩤ (sys.tarFileSystem †
                   {blast (path) ↦ newparent})));
```

rmdirError This function perform error detection.
[rmdir.90] The rmdir() function shall fail if:

EACCES *[rmdir.90.01] Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.*

EBUSY *[rmdir.90.02] The directory to be removed is currently in use by the system or some process and the implementation considers this to be an error.*

EEXIST *or [ENOTEMPTY] [rmdir.90.03] The path argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dotdot.*

EINVAL *[rmdir.90.04] The path argument contains a last component that is dot.*

EIO *[rmdir.90.05] A physical I/O error has occurred.*

ELOOP *[rmdir.90.06] A loop exists in symbolic links encountered during resolution of the path argument.*

ENAMETOOLONG *[rmdir.90.07] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.*

ENOENT *[rmdir.90.08] A component of path does not name an existing file, or the path argument names a nonexistent directory or points to an empty string.*

ENOTDIR *[rmdir.90.10] A component of path is not a directory.*

EPERM *or [EACCES] [rmdir.90.11] The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.*

EROFS *[rmdir.90.12] The directory entry to be removed resides on a read-only file system.*

```
rmdirError : Path × System → char*
rmdirError (unresolvedpath, sys) ≙
  let proc = sys.process (sys.actualproc),
      tar = sys.tarFileSystem,
      c = sys.constants in
  if (pathIsEmpty (unresolvedpath))
  then "[ENOENT]"
  elseif (isLoop (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (symLoop (c ("{SYMLOOP_MAX}")) (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (¬ isValidPath (blast (unresolvedpath), tar))
  then "[ENOTDIR]"
  else let path = resolveLink (unresolvedpath, tar) in
       if (cannotAccess (path, tar, proc.userid, proc.groupid))
       then "[EACCESS]"
       elseif (fileExists (path, tar))
       then "[EEXIST]"
       elseif (isNameTooLong (path, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (isNameTooLong (unresolvedpath, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (prefIsNotDir (path, tar))
       then "[ENOTDIR]"
       elseif (isRdOnly (blast (path), tar, proc.userid, proc.groupid))
       then "[EROFS]"
       else [];
```

[rmdir.91] The rmdir() function may fail if:

ELOOP *[rmdir.91.01] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.*

ENAMETOOLONG *[rmdir.91.02] As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded PATH_MAX.*

2.2.3 Unlink

unlink NAME: unlink – remove a directory entry

```
int unlink(const char * path);
```

[unlink.05] The unlink() function shall remove a link to a file. [unlink.06] If path names a symbolic link, unlink() shall remove the symbolic link named by path and shall not affect any file or directory named by the contents of the symbolic link. [unlink.07] Otherwise, unlink() shall remove the link named by the pathname pointed to by path and shall decrement the link count of the file referenced by the link [unlink.08] When the file's link count becomes 0 and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible [unlink.09] If one or more processes have the file open when the last link is removed, the link shall be removed before unlink() returns, but the removal of the file contents shall be postponed until all references to the file are closed. [unlink.10] The path argument shall not name a directory unless the process has appropriate privileges and the implementation supports using unlink() on directories. [unlink.11] Upon successful completion, unlink() shall mark for update the st_ctime and st_mtime fields of the parent directory. [unlink.12] Also, if the file's link count is not 0, the st_ctime field of the file shall be marked for update. [unlink.13] Upon successful completion, 0 shall be returned. [unlink.14] Otherwise, -1 shall be returned and errno set to indicate the error. [unlink.15] If -1 is returned, the named file shall not be changed.

```
unlink : Path × System → (ℤ × System)
unlink (unresolvedpath, sys) ≜
  let tar = sys.tarFileSystem,
      err = unlinkError (unresolvedpath, sys) in
  if err ≠ []
  then mk- (− 1, μ (sys, errno ↦ err))
  else let path = resolveLink (unresolvedpath, tar),
           parent = tar (blast (path)),
           file = sys.tarFileSystem (path),
           newparentdates = mk- Tar`Dates
                                (
                                 parent.attrib.dates.st-atime,
                                 mk- Tar`Date (parent.attrib.dates.st-ctime.date, true),
                                 mk- Tar`Date (parent.attrib.dates.st-mtime.date, true)),
           pattrib = parent.attrib,
           newparentattrib = mk- Tar`Attributes
                                (
                                 pattrib.type,
                                 pattrib.perm,
                                 pattrib.links − 1,
                                 pattrib.user,
                                 pattrib.group,
                                 pattrib.size − 1,
                                 newparentdates),
           newparent = μ (parent, attrib ↦ newparentattrib) in
       if pathIsSLink (sys, path)
       then mk- (0, μ (sys,

                       tarFileSystem ↦ {unresolvedpath} ⩤ (sys.tarFileSystem †
                        {blast (unresolvedpath) ↦ newparent})))
       elseif file.attrib.links = 1
       then mk- (0, μ (sys,

                       tarFileSystem ↦ {unresolvedpath, path} ⩤ (sys.tarFileSystem †
                        {blast (path) ↦ newparent})))
       else let x1 = μ (sys,

                          tarFileSystem ↦ {unresolvedpath} ⩤ (sys.tarFileSystem †
                           {blast (unresolvedpath) ↦ newparent})),
                newatrib = μ (file.attrib, links ↦ file.attrib.links − 1),
                x2 = μ (file, attrib ↦ newatrib) in
            mk- (0, μ (x1, tarFileSystem ↦ x1.tarFileSystem † {path ↦ x2}));
```

[PathIsSLink] The named file is a symbolic link.

```
pathIsSLink : System × Tar`Path → 𝔹
pathIsSLink (sys, path) ≜
  if path ∉ dom sys.tarFileSystem
  then false
  else let file = sys.tarFileSystem (path) in
       (file.attrib.type = L);
```

unlinkError This function perform error detection.

[unlink.90] The unlink() function shall fail and shall not unlink the file if:

EACCES *[unlink.90.01] Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.*

EBUSY *[unlink.90.02] The file named by the path argument cannot be unlinked because it is being used by the system or another process and the implementation considers this an error.*

ELOOP *[unlink.90.03] A loop exists in symbolic links encountered during resolution of the path argument.*

ENAMETOOLONG *[unlink.90.04] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.*

ENOENT *[unlink.90.05] A component of path does not name an existing file or path is an empty string.*

ENOTDIR *[unlink.90.06] A component of the path prefix is not a directory.*

EPERM *[unlink.90.07] The file named by path is a directory, and either the calling process does not have appropriate privileges, or the implementation prohibits using unlink() on directories.*

EPERM *or [EACCES] [unlink.90.08] The S_ISVTX flag is set on the directory containing the file referred to by the path argument and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.*

EROFS *[unlink.90.09] The directory entry to be unlinked is part of a read-only file system.*

[unlink.92] The unlink() function may fail and not unlink the file if:

ELOOP *[unlink.92.02] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.*

ENAMETOOLONG *[unlink.92.03] As a result of encountering a symbolic link in resolution of the path argument, the length of the substituted pathname string exceeded PATH_MAX.*

```
unlinkError : Path × System → char*
unlinkError (unresolvedpath, sys) △
  let proc = sys.process (sys.actualproc),
      tar = sys.tarFileSystem,
      c = sys.constants in
  if (pathIsEmpty (unresolvedpath))
  then "[ENOENT]"
  elseif (isLoop (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (symLoop (c ("{SYMLOOP_MAX}")) (unresolvedpath, tar))
  then "[ELOOP]"
  elseif (¬ isValidPath (blast (unresolvedpath), tar))
  then "[ENOTDIR]"
  else let path = resolveLink (unresolvedpath, tar) in
       if (cannotAccess (path, tar, proc.userid, proc.groupid))
       then "[EACCESS]"
       elseif (isNameTooLong (path, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (isNameTooLong (unresolvedpath, c ("{PATH_MAX}"), c ("{NAME_MAX}")))
       then "[ENAMETOOLONG]"
       elseif (prefIsNotDir (path, tar))
       then "[ENOTDIR]"
       elseif (isRdOnly (blast (path), tar, proc.userid, proc.groupid))
       then "[EROFS]"
       else [];
```

2.2.4 Remove

remove NAME: remove – remove a file or a directory file

```
int remove(const char *path);
```

[remove.01] The remove() function shall cause the file named by the pathname pointed to by path to be no longer acessible by that name. [remove.02] A subsequent attempt to open that file using that name shall fail, unless it is created a new. If path does not name a directory, remove(path) shall be equivalent to unlink(path). If path names a directory, remove(path) shall be equivalent to rmdir(path).

$$
\begin{array}{l}
remove : Path \times System \rightarrow (\mathbb{Z} \times System) \\
remove\,(unresolvedpath, sys) \triangleq \\
\quad \mathsf{let}\ tar = sys.tarFileSystem\ \mathsf{in} \\
\quad \mathsf{let}\ path = resolveLink\,(unresolvedpath, tar)\ \mathsf{in} \\
\quad \mathsf{if}\ pathIsDirectory\,(sys, path) \\
\quad \mathsf{then}\ rmdir\,(path, sys) \\
\quad \mathsf{else}\ unlink\,(path, sys);
\end{array}
$$

[PathIsDirectory] The named file is a directory.

$$
\begin{array}{l}
pathIsDirectory : System \times Tar`Path \rightarrow \mathbb{B} \\
pathIsDirectory\,(sys, path) \triangleq \\
\quad \mathsf{if}\ path \notin \mathsf{dom}\ sys.tarFileSystem \\
\quad \mathsf{then\ false} \\
\quad \mathsf{else\ let}\ file = sys.tarFileSystem\,(path)\ \mathsf{in} \\
\qquad (file.attrib.type = \textsc{d});
\end{array}
$$

2.2.5 Open

Open *[open.01] The open() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. [open.02.01] The open() function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. [open.02.02] The open file description is new, and therefore the file descriptor shall not share it with any other process in the system. [open.02.03] The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared. [open.03] The file offset used to mark the current position within the file shall be set to the beginning of the file. [open.04] The file status flags and file access modes of the open file description shall be set according to the value of oflag. [app.open.104] Values for oflag are constructed by a bitwise-inclusive OR of flags from the following list, defined in <fcntl.h>.>Applications shall specify exactly one of the first three values (file access modes) below in the value of oflag:O_RDONLY Open for reading only. O_WRONLY Open for writing only. O_RDWR Open for reading and writing. The result is undefined if this flag is applied to a FIFO. [open.05] O_APPEND If set, the file offset shall be set to the end of the file prior to each write. [open.06.01] If the file exists, this flag has no effect except as noted under O_EXCL below. [open.06.02] Otherwise, the file shall be created; [open.06.03] the user ID of the file shall be set to the effective user ID of the process; [open.06.04] the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; [open.06.05] and the access permission bits (see <sys/stat.h>) of the file mode shall be set to the value of the third argument taken as type mode_t modified as follows: a bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. [open.13.01] If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. [open.13.02] It shall have no effect on FIFO special files or terminal device files. [app.open.13.03] Its effect on other file types is implementation-defined. [app.open.13.04] The result of using O_TRUNC with O_RDONLY is undefined. [open.14] If O_CREAT is set and the file did not previously exist, upon successful completion, open() shall mark for update the st_atime, st_ctime, and st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory. [open.15] If O_TRUNC is set and the file did previously exist, upon successful completion, open() shall mark for update the st_ctime and st_mtime fields of the file. [open.30.02] Otherwise, -1 shall be returned and errno set to indicate the error. [open.30.03] No files shall be created or modified if the function returns -1.*

```
open : Tar'Path × Flags-set × Mode-t-set × System → (ℤ × System)
open (unpath, flags, mode-t, sys) △
  let mode = mode2perm (mode-t),
      tar = sys.tarFileSystem,
      proc = sys.process (sys.actualproc),
      err = openError (unpath, flags, sys) in
  if err ≠ []
  then mk-(− 1, μ (sys, errno ↦ err))
  else let path = resolveLink (unpath, sys.tarFileSystem) in
       if (isRDWRFifo (sys, path, flags))
       then mk-(− 1, sys)
       else let ofd = createOfd (sys, path, flags),
                fd = chooseFd (sys) in
            if (O_CREAT ∈ flags ∧ path ∉ dom tar)
            then let dates = mk-Tar'Dates
                                 (
                                  mk-Tar'Date (nil , true),
                                  mk-Tar'Date (nil , true),
                                  mk-Tar'Date (nil , true)),
                     file = mk-Tar'File (mk-Tar'Attributes (R,
                                                            applyMask (mode, proc.umask), 1,
                                                            proc.userid, proc.groupid, 0, dates),
                                         mk-Tar'FileContents ({↦})),
                     pdir = tar (blast (path)),
                     pattrib = pdir.attrib,
                     pdates = pattrib.dates,
                     npattrib = μ (pattrib, dates ↦ mk-Tar'Dates
                                                    (
                                                     pdates.st-atime,
                                                     mk-Tar'Date (pdates.st-ctime.date, true),
                                                     mk-Tar'Date (pdates.st-mtime.date, true))),
                     newpdir = μ (pdir, attrib ↦ npattrib),
                     newproc = μ (proc, usedfds ↦ proc.usedfds ∪ {fd}),
                     newsys = μ (sys, tarFileSystem ↦ tar † {path ↦ file, blast (path) ↦ newpdir},
                                      fds ↦ sys.fds † {fd ↦ ofd},
                                      process ↦ sys.process † {sys.actualproc ↦ newproc}) in
                 mk-(fd, newsys)
            else if (path ∈ dom tar ∧ tar (path).attrib.type = R ∧
                     (O_RDWR ∈ flags ∨ O_WRONLY ∈ flags) ∧
                     O_TRUNC ∈ flags)
                 then let file = tar (path),
                          fattrib = file.attrib,
                          fdates = fattrib.dates,
                          nfattrib = μ (fattrib, size ↦ 0, dates ↦ mk-Tar'Dates
                                                                   (
                                                                    fdates.st-atime,
                                                                    mk-Tar'Date (fdates.st-ctime.date, true),
                                                                    mk-Tar'Date (fdates.st-mtime.date, true))),
                          newfile = μ (file,

                                       attrib ↦ nfattrib,
                                       contents ↦ mk-Tar'FileContents ({↦})),
                          newproc = μ (proc, usedfds ↦ proc.usedfds ∪ {fd}),
                          newsys = μ (sys, tarFileSystem ↦ tar † {path ↦ newfile},
                                           fds ↦ sys.fds † {fd ↦ ofd},
                                           process ↦ sys.process † {sys.actualproc ↦ newproc}) in
                      mk-(fd, newsys)
                 else let newproc = μ (proc, usedfds ↦ proc.usedfds ∪ {fd}),
                          newsys = μ (sys, fds ↦ sys.fds † {fd ↦ createOfd (sys, path, flags)},
                                           process ↦ sys.process † {sys.actualproc ↦ newproc}) in
                      mk-(fd, newsys);
```

[open.02.01] The open() function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. [open.02.02] The open file description is new, and therefore the file descriptor shall not share it with any other process in the system.

```
chooseFd : Tar'System → ℕ
chooseFd (sys) ≙
  let used = usedFds (rng sys.process) in
  chooseFDaux (0, used);

usedFds : Tar'ProcInfo-set → ℕ-set
usedFds (s) ≙
  if s = {}
  then {}
  else let proc ∈ s in
       proc.usedfds ∪ usedFds (s \ {proc});

chooseFDaux : ℕ × ℕ-set → ℕ
chooseFDaux (n, s) ≙
  if n ∉ s
  then n
  else chooseFDaux (n + 1, s);
```

[open.02.03] The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared. [open.03] The file offset used to mark the current position within the file shall be set to the beginning of the file. [open.04] The file status flags and file access modes of the open file description shall be set according to the value of oflag. [open.19] The largest value that can be represented correctly in an object of type off_t shall be established as the offset maximum in the open file description

```
createOfd : Tar'System × Tar'Path × Tar'Flags-set → Tar'OpenFileDescriptor
createOfd (sys, path, flags) ≙
  let ofd = mk-Tar'OpenFileDescriptor (path, {}, 0, sys.constants ("{MAX_OFFSET}"), {}, RD) in
  open04 (ofd, flags);

open04 : Tar'OpenFileDescriptor × Tar'Flags-set → Tar'OpenFileDescriptor
open04 (ofd, flags) ≙
  if O_APPEND ∈ flags
  then open04aux (μ (ofd, status ↦ ofd.status ∪ {O_APPEND}), flags)
  else open04aux (ofd, flags);

open04aux : Tar'OpenFileDescriptor × Tar'Flags-set → Tar'OpenFileDescriptor
open04aux (ofd, flags) ≙
  if O_RDONLY ∈ flags
  then μ (ofd, mode ↦ RD)
  else if O_WRONLY ∈ flags
       then μ (ofd, mode ↦ WR)
       else μ (ofd, mode ↦ RDWR);
```

[app.open.104] The result is undefined if this flag is applied to a FIFO.

```
isRDWRFifo : Tar'System × Tar'Path × Tar'Flags-set → 𝔹
isRDWRFifo (sys, path, flags) ≙
  if path ∈ dom sys.tarFileSystem
  then sys.tarFileSystem (path).attrib.type = P ∧ O_RDWR ∈ flags
  else false;
```

openError *[open.31] The open() function shall fail if:*

ELOOP *[open.31.07] A loop exists in symbolic links encountered during resolution of the path argument.*

ENOTDIR *[open.31.14] A component of the path prefix is not a directory.*

EEXIST *[open.31.02] O_CREAT and O_EXCL are set, and the named file exists.*

EINTR *[open.31.03] A signal was caught during open().*

EISDIR *[open.31.06] The named file is a directory and oflag includes O_WRONLY or O_RDWR.*

EMFILE *[open.31.08] OPEN_MAX file descriptors are currently open in the calling process.*

ENAMETOOLONG *[open.31.09] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.*

ENFILE *[open.31.10] The maximum allowable number of files is currently open in the system.*

ENOENT *[open.31.11] O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the path argument points to an empty string.*

ENOSR *[open.31.12] The path argument names a STREAMS-based file and the system is unable to allocate a STREAM.*

ENOSPC *[open.31.13] The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.*

ENXIO *[open.31.15] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.*

ENXIO *[open.31.16] The named file is a character special or block special file, and the device associated with this special file does not exist.*

EOVERFLOW *[open.31.17] The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.*

EROFS *[open.31.18] The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the oflag argument.*

[open.32] The open() function may fail if:

EINVAL *[open.32.02] The value of the oflag argument is not valid.*

ELOOP *[open.32.03] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument.*

ENOMEM *[open.32.05] The path argument names a STREAMS file and the system is unable to allocate resources.*

```
openError : Tar'Path × Flags-set × System → char*
openError (unpath, flags, sys) △
  let c = sys.constants,
      tar = sys.tarFileSystem in
  if (isLoop (unpath, sys.tarFileSystem))
  then "[ELOOP]"
  elseif (symLoop (c ("{SYMLOOP_MAX}")) (unpath, tar))
  then "[ELOOP]"
  elseif (unpath.path ≠ [] ∧ (¬ isValidPath (blast (unpath), tar)))
  then "[ENOTDIR]"
  else let path = resolveLink (unpath, sys.tarFileSystem) in
       if (open3114 (sys, path))
       then "[ENOTDIR]"
       elseif (open3101 (sys, path, flags))
       then "[EACCESS]"
       elseif (open3102 (sys, path, flags))
       then "[EEXIST]"
       elseif (open3103 (sys))
       then "[EINTR]"
       elseif (open3106 (sys, path, flags))
       then "[EISDIR]"
       elseif (open3108 (sys))
       then "[EMFILE]"
       elseif (open3109 (sys, path))
       then "[ENAMETOOLONG]"
       elseif (open3110 (sys))
       then "[ENFILE]"
       elseif (open3111 (sys, path, flags))
       then "[ENOENT]"
       elseif (open3112 (sys, path))
       then "[ENOSR]"
       elseif (open3113 (sys, path, flags))
       then "[ENOSPC]"
       elseif (open3115 (sys, path, flags))
       then "[ENXIO]"
       elseif (open3116 (sys, path))
       then "[ENXIO]"
       elseif (open3117 (sys, path))
       then "[EOVERFLOW]"
       elseif (open3118 (sys, path, flags))
       then "[EROFS]"
       elseif (open3202 (flags))
       then "[EINVAL]"
       elseif (open3205 (sys, path))
       then "[ENOMEM]"
       else [];
```

[open.31.07] [ELOOP] A loop exists in symbolic links encountered during resolution of the path argument. [open.32.03] [ELOOP] More than SYMLOOP_MAX symbolic links were encountered during resolution of the path argument. These functions are defined in the auxiliary functions

[open.31.14] [ENOTDIR] A component of the path prefix is not a directory.

```
open3114 : Tar'System × Tar'Path → 𝔹
open3114 (sys, path) △
  prefIsNotDir (path, sys.tarFileSystem);
```

[open.31.01] [EACCES] Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by oflag are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or O_TRUNC is specified and write permission is denied.

```
open3101 : System × Tar`Path × Flags-set → B
open3101 (sys, path, flags) △
  let proc = sys.process (sys.actualproc) in
  ∃ p ∈ subpaths (path) ·
      ¬ checkExecPerm (p, sys.tarFileSystem, proc.userid, proc.groupid) ∨
      if (path ∈ dom sys.tarFileSystem)
      then ¬ checkFilePerm (sys, path, flags)
      else ¬ checkWritePerm (blast (path), sys.tarFileSystem, proc.userid, proc.groupid) ∨
           (O_TRUNC ∈ flags∧¬ checkWritePerm (path, sys.tarFileSystem, proc.userid, proc.groupid));
```

[open.31.02] [EEXIST] O_CREAT and O_EXCL are set, and the named file exists.

```
open3102 : System × Tar`Path × Flags-set → B
open3102 (sys, path, flags) △
  O_CREAT ∈ flags ∧
  O_EXCL ∈ flags ∧
  path ∈ dom sys.tarFileSystem;
```

[open.31.03] [EINTR] A signal was caught during open().

```
open3103 : System → B
open3103 (sys) △
  sys.environment.signal;
```

[open.31.06] [EISDIR] The named file is a directory and oflag includes O_WRONLY or O_RDWR.

```
open3106 : System × Tar`Path × Flags-set → B
open3106 (sys, path, flags) △
  if path ∉ dom sys.tarFileSystem
  then false
  else let file = sys.tarFileSystem (path) in
       (file.attrib.type = D) ∧ (O_WRONLY ∈ flags ∨ O_RDWR ∈ flags);
```

[open.31.08] [EMFILE] OPEN_MAX file descriptors are currently open in the calling process.

```
open3108 : System → B
open3108 (sys) △
  let proc = sys.process (sys.actualproc) in
  sys.constants ("{OPEN_MAX}") = card proc.usedfds;
```

[open.31.09] [ENAMETOOLONG] The length of the path argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

```
open3109 : Tar`System × Tar`Path → B
open3109 (sys, path) △
  let rpath = resolveLink (path, sys.tarFileSystem),
      c = sys.constants in
  if path.path = []
  then false
  else isNameTooLong (rpath, c ("{PATH_MAX}"), c ("{NAME_MAX}"))
pre if (path.path ≠ [])
    then isValidPath (blast (path), sys.tarFileSystem)
    else true
  ;
```

[open.31.10] [ENFILE] The maximum allowable number of files is currently open in the system.

```
open3110 : System → B
open3110 (sys) △
  let c = sys.constants in
  card usedFds (rng sys.process) = c ("{MAX_OPEN_SYSTEM}");
```

[open.31.11] [ENOENT] O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the path argument points to an empty string.

$$
\begin{array}{l}
open3111 : System \times Tar`Path \times Flags\text{-set} \to \mathbb{B} \\
open3111\,(sys, path, flags) \triangleq \\
\quad (\text{O\_CREAT} \notin flags \wedge path \notin \mathsf{dom}\ sys.tarFileSystem) \vee \\
\quad (\text{O\_CREAT} \in flags \wedge (path.path = [] \vee (blast\,(path) \notin \mathsf{dom}\ sys.tarFileSystem)));
\end{array}
$$

[open.31.12] [ENOSR] The path argument names a STREAMS-based file and the system is unable to allocate a STREAM.

$$
\begin{array}{l}
open3112 : Tar`System \times Tar`Path \to \mathbb{B} \\
open3112\,(sys, path) \triangleq \\
\quad \mathsf{if}\ path \in \mathsf{dom}\ sys.tarFileSystem \\
\quad \mathsf{then}\ sys.tarFileSystem\,(path).attrib.type = \text{S} \wedge sys.environment.unable2socket \\
\quad \mathsf{else}\ \mathsf{false};
\end{array}
$$

[open.31.13] [ENOSPC] The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.

$$
\begin{array}{l}
open3113 : Tar`System \times Tar`Path \times Flags\text{-set} \to \mathbb{B} \\
open3113\,(sys, path, flags) \triangleq \\
\quad sys.unusedspace = 0 \vee \\
\quad \mathsf{if}\ path.path \neq [] \\
\quad \mathsf{then}\ (path \notin \mathsf{dom}\ sys.tarFileSystem \wedge \\
\qquad \text{O\_CREAT} \in flags \wedge \\
\qquad isLinkFull\,(path, sys.tarFileSystem, sys.constants\,(\text{"}\{LINK\_MAX\}\text{"}))) \\
\quad \mathsf{else}\ \mathsf{false} \\
\mathsf{pre}\ \ \mathsf{if}\ path.path = [] \\
\quad \mathsf{then}\ \mathsf{true} \\
\quad \mathsf{else}\ blast\,(path) \in \mathsf{dom}\ sys.tarFileSystem \\
\quad ;
\end{array}
$$

[open.31.15] [ENXIO] O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.

$$
\begin{array}{l}
open3115 : Tar`System \times Tar`Path \times Flags\text{-set} \to \mathbb{B} \\
open3115\,(sys, path, flags) \triangleq \\
\quad \text{O\_NONBLOCK} \in flags \wedge \\
\quad sys.tarFileSystem\,(path).attrib.type = \text{P} \wedge \\
\quad \text{O\_WRONLY} \in flags \wedge \\
\quad \neg\, \exists\, fd \in \mathsf{rng}\ sys.fds \cdot fd.file = path \wedge fd.mode = \text{RD};
\end{array}
$$

[open.31.16] [ENXIO] The named file is a character special or block special file, and the device associated with this special file does not exist.

$$
\begin{array}{l}
open3116 : Tar`System \times Tar`Path \to \mathbb{B} \\
open3116\,(sys, path) \triangleq \\
\quad \mathsf{if}\ path \in \mathsf{dom}\ sys.tarFileSystem \\
\quad \mathsf{then\ let}\ file = sys.tarFileSystem\,(path)\ \mathsf{in} \\
\qquad \mathsf{if}\ (file.attrib.type = \text{C} \vee file.attrib.type = \text{B}) \\
\qquad \mathsf{then}\ file.contents \notin \mathsf{dom}\ sys.devices \\
\qquad \mathsf{else}\ \mathsf{false} \\
\quad \mathsf{else}\ \mathsf{false};
\end{array}
$$

[open.31.17] [EOVERFLOW] The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.

$$
\begin{array}{l}
open3117 : Tar`System \times Tar`Path \to \mathbb{B} \\
open3117\,(sys, path) \triangleq \\
\quad \mathsf{if}\ path \in \mathsf{dom}\ sys.tarFileSystem \\
\quad \mathsf{then\ let}\ file = sys.tarFileSystem\,(path)\ \mathsf{in} \\
\qquad file.attrib.type = \text{R} \wedge file.attrib.size > sys.constants\,(\text{"}\{MAX\_OFF\_T\}\text{"}) \\
\quad \mathsf{else}\ \mathsf{false};
\end{array}
$$

[open.31.18] [EROFS] The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if the file does not exist), or O_TRUNC is set in the oflag argument.

$$
\begin{array}{l}
open3118 : Tar`System \times Tar`Path \times Tar`Flags\text{-}\mathsf{set} \rightarrow \mathbb{B} \\
open3118\,(sys, path, flags) \triangleq \\
\quad (\text{O\_WRONLY} \in flags \lor \\
\quad \text{O\_RDWR} \in flags \lor \\
\quad (\text{O\_CREAT} \in flags \land path \notin \mathsf{dom}\ sys.tarFileSystem) \lor \\
\quad \text{O\_TRUNC} \in flags) \land \\
\quad sys.environment.fsmode = \text{RD};
\end{array}
$$

[open.32.02] [EINVAL] The value of the oflag argument is not valid.

$$
\begin{array}{l}
open3202 : Tar`Flags\text{-}\mathsf{set} \rightarrow \mathbb{B} \\
open3202\,(flags) \triangleq \\
\quad \mathsf{card}\ \{x \mid x \in flags \cdot x = \text{O\_RDONLY} \lor x = \text{O\_WRONLY} \lor x = \text{O\_RDWR}\} > 1 \lor \\
\quad (\text{O\_EXCL} \in flags \land \text{O\_CREAT} \notin flags) \lor \\
\quad (\{\text{O\_TRUNC}, \text{O\_RDONLY}\} \subseteq flags);
\end{array}
$$

[open.32.05] [ENOMEM] The path argument names a STREAMS file and the system is unable to allocate resources.

$$
\begin{array}{l}
open3205 : Tar`System \times Tar`Path \rightarrow \mathbb{B} \\
open3205\,(sys, path) \triangleq \\
\quad \mathsf{if}\ path \in \mathsf{dom}\ sys.tarFileSystem \\
\quad \mathsf{then}\ sys.tarFileSystem\,(path).attrib.type = \text{S} \land sys.environment.unable2resorces \\
\quad \mathsf{else}\ \mathsf{false};
\end{array}
$$

2.2.6 Read

read read input

[read.01] The read() function shall attempt to read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf. [read.03] Before any action described below is taken, and if nbyte is zero, the read() function may detect and return errors as described below. [read.04] In the absence of errors, or if error detection is not performed, the read() function shall return zero and have no other results.

[read.05] On files that support seeking (for example, a regular file), the read() shall start at a position in the file given by the file offset associated with fildes. [read.06] The file offset shall be incremented by the number of bytes actually read. [read.07] Files that do not support seeking - for example, terminals - always read from the current position. [read.08] If the starting position is at or after the end-of-file, 0 shall be returned. [app.read.51] If the value of nbyte is greater than SSIZE_MAX, the result is implementation-defined.

OBS: The file types that support or don't seeking are not mentioned. We consider that only the regular files support seeking. This is not an important aspect because the function is prepared only to deal with regular files.

[app.read.51] If the value of nbyte is greater than SSIZE_MAX, the result is implementation-defined.

OBS: We choose not to read any byte if this condition is true.

[read.13.01] Upon successful completion, where nbyte is greater than 0, read() shall mark for update the st_atime field of the file [read.13.02] and shall return the number of bytes read. This number shall never be greater than nbyte. [read.14] The value returned may be less than nbyte [read.14.01] if the number of bytes left in the file is less than nbyte, [read.41.01] Upon successful completion, read() and pread() shall return a non-negative integer indicating the number of bytes actually read. [read.85] The minimum of either the number of bytes requested or the number

of bytes currently available shall be returned without waiting for more bytes to be input. If no characters are available, read() shall return a value of zero, having read no data.

OBS: Doesn't mention if the return value must be the number of red bytes or the number of bytes to be put on buf. These values could be different so we choose to return the number of bytes red from the file.

OBS: Acording to [read.68.01], there is one more situation in which the return value is less then nbyte.

```
read : System × ℕ × ℕ × ℕ → ℤ × System
read (sys, filedes, buf, nbyte) △
  let err = readError (sys, filedes, nbyte) in
  if err ≠ []
  then mk- (− 1, μ (sys, errno ↦ err))
  else let fileContents = getFileContents (sys, filedes),
           file = getFile (sys, filedes),
           fd = sys.fds (filedes),
           offset = getOffset (sys, filedes),
           maxoffset = getMaxOffset (sys, filedes),
           constants = sys.constants,
           flags = fd.flags in
       if nbyte = 0
       then mk- (0, sys)
       else let startd = if isRegularFile (file)
                         then offset
                         else 0 in
            if startAtEOF (fileContents, startd)
            then mk- (0, sys)
            elseif gNByte (constants, filedes)
            then mk- (0, sys)
            else let mk- (data, newoffset) = readFile (fileContents, flags, startd, nbyte, maxoffset) ([]),
                     bytesRead = newoffset − offset,
                     sys1 = setOffset (sys, filedes, newoffset),
                     sys2 = setSt-ATime (sys1, filedes),
                     res = bytesRead,
                     sys3 = setMemory (sys2, buf, data) in
                 mk- (res, sys3)
pre getFile (sys, filedes).attrib.type = R
  ;
```

This function verify if a file is regular.

```
isRegularFile : File → ℬ
isRegularFile (file) △
  file.attrib.type = R;
```

This function verify if the starting position is at or after the end-of-file.

```
startAtEOF : FileContents × ℕ → ℬ
startAtEOF (filecont, startd) △
  startd ≥ max (dom filecont.content);
```

This function verify if the value of nbyte is greater than SSIZE_MAX.

```
gNByte : Constants × ℕ → ℬ
gNByte (consts, nbyte) △
  nbyte > consts ("SSIZE_MAX");
```

readError performs error detection

[read.42] The read() and pread() functions shall fail if:

EBADF *[read.42.02] The fildes argument is not a valid file descriptor open for reading.*

EISDIR *[read.42.07] The fildes argument refers to a directory and the implementation does not allow the directory to be read using read() or pread(). The readdir() function should be used instead.*

EOVERFLOW *[read.42.08] The file is a regular file, nbyte is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with fildes.*

[read.44] The read() and pread() functions may fail if:

EIO *[read.44.01] A physical I/O error has occurred.*

This function performs error detection.

```
readError : System × ℕ × ℕ → char*
readError (sys, filedes, nbyte) △
  if filedes ∉ dom sys.fds
  then "[EBADFS]"
  else let file = getFile (sys, filedes),
           fd = sys.fds (filedes) in
       if notValidFDR (sys.tarFileSystem, sys.fds (filedes), file, sys.process (sys.actualproc))
       then "[EBADF]"
       else if isDirectory (file)
            then "[EISDIR]"
            elseif file.attrib.type = R ∧ nbyte > 0 ∧ geMaxOffset (fd)
            then "[EOVERFLOW]"
            elseif physicalError (sys)
            then "[EIO]"
            else [];
```

This function verifies if the file descriptor is a valid file descriptor open for reading.

```
notValidFDR : Tar × OpenFileDescriptor × File × ProcInfo → 𝔹
notValidFDR (fs, fd, file, pi) △
  if fd.file ∉ dom fs
  then true
  elseif fd.mode = WR
  then true
  elseif file.attrib.user = pi.userid
  then ¬ file.attrib.perm.user.read
  elseif file.attrib.group = pi.groupid
  then ¬ file.attrib.perm.group.read
  else ¬ file.attrib.perm.other.read;
```

This function verifies if the file descriptor refers to a directory.

```
isDirectory : File → 𝔹
isDirectory (file) △
  file.attrib.type = D;
```

This functions verifies if the starting position is greater than or equal to the offset maximum established in the open file descriptor.

```
geMaxOffset : OpenFileDescriptor → 𝔹
geMaxOffset (ofd) △
  ofd.offset > ofd.maxoffset;
```

This function verify if there is a physical error.

```
physicalError : System → 𝔹
physicalError (sys) △
  sys.environment.physicalerror;
```

readFile read contents from file

[read.17] For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with fildes. [read.11] If any portion of a regular file prior to the end-of-file has not been written, read() shall return bytes with value 0.

[read.61.02] If ISIG is set, the INTR character shall be discarded when processed. [read.62.02] If ISIG is set, the QUIT character shall be discarded when processed.

[read.66.01] Special character on input, which is recognized if the ICANON flag is set. Erases the last character in the current line; see Canonical Mode Input Processing. It shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. [read.66.02] If ICANON is set, the ERASE character shall be discarded when processed.

[&&read.66.01] The `ERASE` character shall delete the last character in the current line, if there is one. [&&read.67.01] The `KILL` character shall delete all data in the current line, if there is any. [&read.66.01] The `ERASE` and `KILL` characters shall have no effect if there is no data in the current line. [&read.67.01] The `ERASE` and `KILL` characters shall have no effect if there is no data in the current line.

OBS: These last requirements are similar to [read.66.01] and [read.67.01]. [&read.66.01] and [&read67.01] are implemented by [read.66.01] and [read.67.01]. The actions described in [&&read.66.01] and [&&read.67.01] only ocour if an aditional condition is verified. We choose to mantain the requirements that have stronger conditions ([read.66.01] and [read.67.01]).

[read.67.01] Special character on input, which is recognized if the ICANON flag is set. Deletes the entire line, as delimited by an NL, EOF, or EOL character. [&read.67.02] If ICANON is set, the KILL character shall be discarded when processed.

[read.68.01] Special character on input, which is recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process without waiting for a newline, and the EOF is discarded. Thus, if there are no bytes waiting (that is, the EOF occurred at the beginning of a line), a byte count of zero shall be returned from the read(), representing an end-of-file indication. [read.68.02] If ICANON is set, the EOF character shall be discarded when processed.

[read.63.02] The SUSP character shall be discarded when processed.

[&&read.64.02] If IXON is set, the STOP character shall be discarded when processed.

[read.64.02] When IXON is not set, the START and STOP characters shall be read. [&&read.65.02] If IXON is set, the START character shall be discarded when processed. [read.65.02] When IXON is not set, the START and STOP characters shall be read.

[read.64.02] [read.65.02] When IXON is not set, the START and STOP characters shall be read.

OBS: This requirement is already implemented by [&&read.64.02] and [&&read.65.02]

[read.71.01] Special character on input, which is recognized if the ICANON flag is set; it is the carriage-return character. [&read.71.02] When ICANON and ICRNL are set and IGNCR is not set, this character shall be translated into an NL, and shall have the same effect as an NL character. [read.76.02] If IGNCR is set, a received CR character shall be ignored (not read). [read.76.01] If INLCR is set, a received NL character shall be translated into a CR character.

[read.71.02] If IGNCR is not set and ICRNL is set, a received CR character shall be translated into an NL character.

OBS: This requirement is similar to [&read.71.02], but it needs one more condition, so we decided to implement the last one.

This function read nbyte bytes from the file associated with the open file descriptor, fildes, into the buffer pointed to by buf.

```
readFile : FileContents × FDFlags-set × ℕ × ℕ × ℕ → Byte* → Byte* × ℕ
readFile (file, flags, pos, maxbytes, maxpos)(ac) △
  let mk-FileContents (conts) = file,
      mk- (last, -) = lastCont (file) in
  if maxbytes = 0 ∨ pos > last ∨
     pos > maxpos
  then mk- (ac, pos)
  else if pos ∉ dom conts
       then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [mk-token (0)])
       else let x = conts (pos) in
            if x = mk-token ("INTR")
            then if isFlagSet (flags, ISIG)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("QUIT")
            then if isFlagSet (flags, ISIG)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("ERASE")
            then if isFlagSet (flags, ICANON)
                 then if len ac > 0 ∧ ac (len ac) ∉ {mk-token ("NL"),
                                                     mk-token ("EOF"),
                                                     mk-token ("EOL")}
                      then readFile (file, flags, pos + 1, maxbytes − 1, maxpos)
                                    (
                                     [ac (i) | i ∈ inds ac · i < len ac])
                      else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("KILL")
            then if isFlagSet (flags, ICANON)
                 then if len ac > 0 ∧ ac (len ac) ∉ {mk-token ("NL"),
                                                     mk-token ("EOF"),
                                                     mk-token ("EOL")}
                      then readFile (file, flags, pos, maxbytes, maxpos)
                                    (
                                     [ac (i) | i ∈ inds ac · i < len ac])
                      else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("EOF")
            then if isFlagSet (flags, ICANON)
                 then mk- (ac, pos + 1)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("SUSP")
            then if isFlagSet (flags, ISIG)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("STOP")
            then if isFlagSet (flags, IXON)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("START")
            then if isFlagSet (flags, IXON)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("CR")
            then if isFlagSet (flags, ICANON) ∧ isFlagSet (flags, ICRNL) ∧ ¬ isFlagSet (flags, IGNCR)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [mk-token ("NL")])
                 elseif isFlagSet (flags, IGNCR)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac)
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            elseif x = mk-token ("NL")
            then if isFlagSet (flags, INLCR)
                 then readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [mk-token ("CR")])
                 else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)])
            else readFile (file, flags, pos + 1, maxbytes − 1, maxpos) (ac ↷ [conts (pos)]);
```

This function verify if flag is set.

$$isFlagSet : FDFlags\text{-set} \times FDFlags \rightarrow \mathbb{B}$$
$$isFlagSet\ (flags, flag) \triangleq$$
$$\quad flag \in flags;$$

2.2.7 Write

write

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

[write.01] The write() function shall attempt to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes.

[write.02.01] Before any action described below is taken, and if nbyte is zero and the file is a regular file, the write() function may detect and return errors as described below. [write.02.02] In the absence of errors, or if error detection is not performed, the write() function shall return zero and have no other results.

[app.write.03] If nbyte is zero and the file is not a regular file, the results are unspecified.

[write.08.01] If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write [write.08.02] no intervening file modification operation shall occur between changing the file offset and the write operation.

[write.04] On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with fildes. [write.06] On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

[write.05] Before successful return from write(), the file offset shall be incremented by the number of bytes actually written.

[write.09] If a write() requests that more bytes be written than there is room for (for example, the process' file size limit or the physical end of a medium), only as many bytes as there is room for shall be written.

```
write : System × ℕ × ℕ × ℕ → (ℤ × System)
write (sys, fildes, buf, nbyte) △
  let ofd = sys.fds (fildes) in
  if (ofd.file ∉ dom sys.tarFileSystem)
  then mk- (nbyte, sys)
  else let err = writeError (sys, fildes, nbyte) in
       if err ≠ []
       then mk- (− 1, μ (sys, errno ↦ err))
       elseif (nbyte = 0)
       then mk- (0, sys)
       elseif (sys.tarFileSystem (ofd.file).attrib.type = R)
       then let file = sys.tarFileSystem (ofd.file),
                attrib = file.attrib in
            if (O_APPEND ∈ ofd.status)
            then if (sys.unusedspace ≥ nbyte)
                 then let newattrib = μ (attrib,
                                         size ↦ attrib.size + nbyte),
                          fcont = mk-Tar'FileContents (file.contents.content †
prepwrite (sys, buf, nbyte, file.attrib.size)),
                          newfile = μ (file, attrib ↦ newattrib, contents ↦ fcont),
                          newsys = μ (sys, unusedspace ↦ (sys.unusedspace −
nbyte), tarFileSystem ↦ sys.tarFileSystem † {ofd.file ↦ newfile}) in
                      mk- (nbyte, setOffset (newsys, fildes, newfile.attrib.size))
                 else let newattrib = μ (attrib, size ↦ attrib.size + sys.unusedspace),
                          fcont = mk-Tar'FileContents (file.contents.content †
prepwrite (sys, buf, sys.unusedspace, file.attrib.size)),
                          newfile = μ (file, attrib ↦ newattrib, contents ↦ fcont),
                          newsys = μ (sys, unusedspace ↦ 0, tarFileSystem ↦ sys.tarFileSystem †
{ofd.file ↦ newfile}) in
                      mk- (sys.unusedspace, setOffset (newsys, fildes, newfile.attrib.size))
            else let file = sys.tarFileSystem (ofd.file) in
                 if ofd.offset + nbyte > file.attrib.size
                 then let diff = ofd.offset + nbyte − file.attrib.size in
                      if (sys.unusedspace > diff)
                      then let newattrib = μ (attrib, size ↦ attrib.size + diff),
                               fcont = mk-Tar'FileContents
                                       (
                                        file.contents.content †
prepwrite (sys, buf, nbyte, ofd.offset)),
                               newfile = μ (file, attrib ↦ newattrib, contents ↦ fcont),
                               newsys = μ (sys, unusedspace ↦ (sys.unusedspace − diff),
                                           tarFileSystem ↦ sys.tarFileSystem † {ofd.file ↦
newfile}) in
                           mk- (nbyte, setOffset (newsys, fildes, newfile.attrib.size))
                      else let newattrib = μ (attrib, size ↦ attrib.size + sys.unusedspace),
                               fcont = mk-Tar'FileContents (file.contents.content †
prepwrite (sys, buf, sys.unusedspace, ofd.offset)),
                               newfile = μ (file, attrib ↦ newattrib, contents ↦ fcont),
                               newsys = μ (sys, unusedspace ↦ 0,
                                           tarFileSystem ↦ sys.tarFileSystem † {ofd.file ↦
newfile}) in
                           mk- (sys.unusedspace, setOffset (newsys, fildes, newfile.attrib.size))
                 else let newattrib = μ (attrib, size ↦ attrib.size + nbyte),
                          fcont = mk-Tar'FileContents (file.contents.content †
prepwrite (sys, buf, nbyte, ofd.offset)),
                          newfile = μ (file, attrib ↦ newattrib, contents ↦ fcont),
                          newsys = μ (sys, tarFileSystem ↦ sys.tarFileSystem † {ofd.file ↦
newfile}) in
                      mk- (nbyte, setOffset (newsys, fildes, ofd.offset + nbyte))
       else mk- (0, sys);
```

Auxiliary functions Used to create the segment of memory that will written in the file contents.

```
prepwrite : Tar`System × ℕ × ℕ × ℕ → Tar`Memory
prepwrite (sys, buf, nbyte, offset) △
   let addrs = mkaddrlist (buf, buf + nbyte),
       fileaddr = mkaddrlist (offset, offset + nbyte),
       array = prepwriteaux (sys, addrs) in
   addrArray2Memory (fileaddr, array);

addrArray2Memory : ℕ* × Tar`Byte* → Tar`Memory
addrArray2Memory (addrs, array) △
   if addrs = []
   then {↦}
   else {hd addrs ↦ hd array} † addrArray2Memory (tl addrs, tl array)
pre len array ≥ len addrs
   ;

prepwriteaux : Tar`System × ℕ* → Tar`Byte*
prepwriteaux (sys, l) △
   cases l :
      [] → [],
      [h] ⌢ t → if (h ∈ dom sys.memory)
                then [sys.memory (h)] ⌢ prepwriteaux (sys, t)
                else [mk-token ("TRASH")] ⌢ prepwriteaux (sys, t)
   end;

mkaddrlist : ℕ × ℕ → ℕ*
mkaddrlist (b, n) △
   if b = n
   then []
   else [b] ⌢ mkaddrlist (b + 1, n)
pre n ≥ b
   ;
```

writeError This function perform error detection
[write.41] The write() and pwrite() functions shall fail if:

EBADF *[write.41.02] The fildes argument is not a valid file descriptor open for writing.*

EFBIG *[write.41.04] The file is a regular file, nbyte is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with fildes.*

EINTR *[write.41.05] The write operation was terminated due to the receipt of a signal, and no data was transferred.*

ENOSPC *[write.41.07] There was no free space remaining on the device containing the file.*

[write.43] The write() and pwrite() functions may fail if:

EIO *[write.43.02] A physical I/O error has occurred.*

ENOBUFS *[write.43.03] Insufficient resources were available in the system to perform the operation.*

[write.44] The write() function may fail if:

EACCES *[write.44.01] A write was attempted on a socket and the calling process does not have appropriate privileges.*

```
writeError : System × ℕ × ℕ → char*
writeError (sys, fildes, nbyte) △
  if (write4102 (sys, fildes))
  then "[EBADF]"
  elseif (write4104 (sys, fildes, nbyte))
  then "[EFBIG]"
  elseif (write4105 (sys))
  then "[EINTR]"
  elseif (write4107 (sys, fildes, nbyte))
  then "[ENOSPC]"
  elseif (write4302 (sys))
  then "[EIO]"
  elseif (write4303 (sys))
  then "[ENOBUFS]"
  elseif (write4401 (sys, fildes))
  then "[EACCES]"
  else [];
```

[write.41.02] [EBADF] The fildes argument is not a valid file descriptor open for writing.

```
write4102 : Tar'System × ℕ → 𝔹
write4102 (sys, fd) △
  fd ∉ dom sys.fds ∨
  if (fd ∈ dom sys.fds)
  then sys.fds (fd).mode = RD
  else false;
```

[write.41.04] [EFBIG] The file is a regular file, nbyte is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with fildes.

```
write4104 : Tar'System × ℕ × ℕ → 𝔹
write4104 (sys, fildes, nbyte) △
  let ofd = sys.fds (fildes) in
  sys.tarFileSystem (ofd.file).attrib.type = R ∧
  nbyte > 0 ∧
  ofd.offset ≥ ofd.maxoffset
pre fildes ∈ dom sys.fds
  ;
```

[write.41.05] [EINTR] The write operation was terminated due to the receipt of a signal, and no data was transferred.

```
write4105 : Tar'System → 𝔹
write4105 (sys) △
  sys.environment.signal;
```

[write.41.07] [ENOSPC] There was no free space remaining on the device containing the file.

```
write4107 : Tar'System × ℕ × ℕ → 𝔹
write4107 (sys, fildes, nbyte) △
  let ofd = sys.fds (fildes) in
  if (O_APPEND ∈ ofd.status ∧ nbyte > 0)
  then sys.unusedspace = 0
  elseif ofd.offset + nbyte > sys.tarFileSystem (ofd.file).attrib.size
  then sys.unusedspace = 0
  else false
pre fildes ∈ dom sys.fds ∧
    sys.fds (fildes).file ∈ dom sys.tarFileSystem
  ;
```

[write.41.08] [EPIPE] An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal shall also be sent to the thread.

```
write4108 : Tar`System × ℕ → 𝔹
write4108 (sys, fildes) △
   let ofd = sys.fds (fildes) in
   sys.tarFileSystem (ofd.file).attrib.type = P ∧
   ¬ ∃ ofds ∈ rng sys.fds · ofds.file = ofd.file ∧ ofds.mode = RD
pre fildes ∈ dom sys.fds
   ;
```

[write.43.02] [EIO] A physical I/O error has occurred.

```
write4302 : Tar`System → 𝔹
write4302 (sys) △
   sys.environment.physicalerror;
```

[write.43.03] [ENOBUFS] Insufficient resources were available in the system to perform the operation.

```
write4303 : Tar`System → 𝔹
write4303 (sys) △
   sys.environment.unable2resorces;
```

[write.44.01] [EACCES] A write was attempted on a socket and the calling process does not have appropriate privileges.

```
write4401 : Tar`System × ℕ → 𝔹
write4401 (sys, fildes) △
   let path = sys.fds (fildes).file in
   (sys.tarFileSystem (path).attrib.type = S ∧
    let proc = sys.process (sys.actualproc) in
    checkWritePerm (path, sys.tarFileSystem, proc.userid, proc.groupid))
pre fildes ∈ dom sys.fds
   ;
```

2.3 Auxiliary Functions

2.3.1 Init

This function produces a value of the type System and it's used to initialize the Tar Object.

```
init : () → System
init () △
   let contents = mk-Tar`FileContents ({0 ↦ 'a'}),
       permissions = mk-Tar`FilePermissions
                          (
                           mk-Tar`Permissions (true, true, true), mk-Tar`Permissions (true, true, true), mk-Tar`Permissions (true,
       fileattrib = mk-Tar`Attributes (R, permissions, 1, 1, 7, 1, mk-Tar`Dates (mk-Tar`Date (mk-token ("1999"), false), mk-Tar`Date (
       file = mk-Tar`File (fileattrib, contents),
       dir = mk-Tar`File (mk-Tar`Attributes (D, permissions, 0, 1, 7, 0, mk-Tar`Dates (mk-Tar`Date (mk-token ("1999"), false), mk-T
       tar = {mk-Tar`Path ([]) ↦ dir,
              mk-Tar`Path (["Desktop"]) ↦ dir,
              mk-Tar`Path (["Desktop", ".DS_Store"]) ↦ file,
              mk-Tar`Path (["Desktop", "CV_sergio.pdf"]) ↦ file},
       proc = {1 ↦ mk-Tar`ProcInfo (1, 7, permissions, {0})},
       ofd = mk-Tar`OpenFileDescriptor (mk-Tar`Path (["Desktop", "CV_sergio.pdf"]), {O_APPEND}, 0, 99999999, {}, RDWR),
       constants = {"{SYMLOOP_MAX}" ↦ 20, "{LINK_MAX}" ↦ 10, "{PATH_MAX}" ↦
20, "{NAME_MAX}" ↦ 256, "{MAX_OFFSET}" ↦ 1024,
                    "{MAX_OPEN_SYSTEM}" ↦ 20, "{MAX_OFF_T}" ↦
2048, "{OPEN_MAX}" ↦ 5, "SSIZE_MAX" ↦ 10} in
   mk-Tar`System (tar, proc, {0 ↦ ofd}, 1, constants, [], 100000, mk-Tar`Environment (false, false, false, false, RDWR, false),
                  {1 ↦ 'h', 2 ↦ 'e', 3 ↦ 'l', 4 ↦ 'l', 5 ↦ 'o'}, {↦});
```

2.3.2 mode2perm

This function transforms the mode_t flags to the correspondent FilePermissions

$mode2perm : Mode\text{-}t\text{-}\mathsf{set} \rightarrow FilePermissions$
$mode2perm\,(smode) \triangleq$
  mk-$FilePermissions\,(usermode\,(smode), groupmode\,(smode), othersmode\,(smode))$;

$usermode : Mode\text{-}t\text{-}\mathsf{set} \rightarrow Permissions$
$usermode\,(modes) \triangleq$
  if $(\text{S\_IRWXU} \in modes)$
  then mk-$Permissions\,(\mathsf{true}, \mathsf{true}, \mathsf{true})$
  else mk-$Permissions\,(\text{S\_IRUSR} \in modes, \text{S\_IWUSR} \in modes, \text{S\_IXUSR} \in modes)$;

$groupmode : Mode\text{-}t\text{-}\mathsf{set} \rightarrow Permissions$
$groupmode\,(modes) \triangleq$
  if $(\text{S\_IRWXG} \in modes)$
  then mk-$Permissions\,(\mathsf{true}, \mathsf{true}, \mathsf{true})$
  else mk-$Permissions\,(\text{S\_IRGRP} \in modes, \text{S\_IWGRP} \in modes, \text{S\_IXGRP} \in modes)$;

$othersmode : Mode\text{-}t\text{-}\mathsf{set} \rightarrow Permissions$
$othersmode\,(modes) \triangleq$
  if $(\text{S\_IRWXO} \in modes)$
  then mk-$Permissions\,(\mathsf{true}, \mathsf{true}, \mathsf{true})$
  else mk-$Permissions\,(\text{S\_IROTH} \in modes, \text{S\_IWOTH} \in modes, \text{S\_IXOTH} \in modes)$;

2.3.3 Various

These are some needed, but self-explaining functions

```
take : ℤ × Tar`Path → Tar`Path
take (n, p) △
   mk-Tar`Path ([p.path (i) | i ∈ inds p.path · i ≤ n])
pre  n ≤ len p.path
   ;

subpaths : Tar`Path → Tar`Path-set
subpaths (p) △
   {take (n − 1, p) | n ∈ inds p.path · n > 0};

blast : Tar`Path → Tar`Path
blast (p) △
   take ((len p.path) − 1, p)
pre  p.path ≠ []
   ;

last : Tar`Path → Tar`Id
last (mk-Tar`Path (p)) △
   p ((len p))
pre  p ≠ []
   ;

drop : ℤ × Tar`Path → Tar`Path
drop (n, p) △
   mk-Tar`Path ([p.path (i) | i ∈ inds p.path · i > n]);

lastCont : FileContents → ℕ × Byte
lastCont (fc) △
   let pos = max (dom fc.content) in
   mk- (pos, fc.content (pos));

max : ℕ-set → ℕ
max (s) △
   if card s = 1
   then let {x} = s in
        x
   else let x ∈ s in
        if x > max (s \ {x})
        then x
        else max (s \ {x})
pre  card s > 0
   ;
```

2.3.4 checkExecPerm

Verifies if a process has search/execution permission on a file

$$
\begin{array}{l}
checkExecPerm : Tar`Path \times Tar`Tar \times Tar`UserId \times Tar`GroupId \rightarrow \mathbb{B} \\
checkExecPerm\,(path, tar, uid, gid) \triangleq \\
\quad userExecPerm\,(path, tar, uid) \vee groupExecPerm\,(path, tar, gid) \vee otherExecPerm\,(path, tar) \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
userExecPerm : Tar`Path \times Tar`Tar \times Tar`UserId \rightarrow \mathbb{B} \\
userExecPerm\,(path, tar, uid) \triangleq \\
\quad \textsf{if}\ (tar\,(path).attrib.user = uid) \\
\quad \textsf{then}\ tar\,(path).attrib.perm.user.exec \\
\quad \textsf{else false} \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
groupExecPerm : Tar`Path \times Tar`Tar \times Tar`GroupId \rightarrow \mathbb{B} \\
groupExecPerm\,(path, tar, gid) \triangleq \\
\quad \textsf{if}\ (tar\,(path).attrib.group = gid) \\
\quad \textsf{then}\ tar\,(path).attrib.perm.group.exec \\
\quad \textsf{else false} \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
otherExecPerm : Tar`Path \times Tar`Tar \rightarrow \mathbb{B} \\
otherExecPerm\,(path, tar) \triangleq \\
\quad tar\,(path).attrib.perm.other.exec \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ;
\end{array}
$$

2.3.5 checkWritePerm

Verifies if a process has write permission on a file

$$
\begin{array}{l}
checkWritePerm : Tar`Path \times Tar`Tar \times Tar`UserId \times Tar`GroupId \rightarrow \mathbb{B} \\
checkWritePerm\,(path, tar, uid, gid) \triangleq \\
\quad userWritePerm\,(path, tar, uid) \vee groupWritePerm\,(path, tar, gid) \vee otherWritePerm\,(path, tar) \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
userWritePerm : Tar`Path \times Tar`Tar \times Tar`UserId \rightarrow \mathbb{B} \\
userWritePerm\,(path, tar, uid) \triangleq \\
\quad \textsf{if}\ (tar\,(path).attrib.user = uid) \\
\quad \textsf{then}\ tar\,(path).attrib.perm.user.write \\
\quad \textsf{else false} \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
groupWritePerm : Tar`Path \times Tar`Tar \times Tar`GroupId \rightarrow \mathbb{B} \\
groupWritePerm\,(path, tar, gid) \triangleq \\
\quad \textsf{if}\ (tar\,(path).attrib.group = gid) \\
\quad \textsf{then}\ tar\,(path).attrib.perm.group.write \\
\quad \textsf{else false} \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ; \\
\\
otherWritePerm : Tar`Path \times Tar`Tar \rightarrow \mathbb{B} \\
otherWritePerm\,(path, tar) \triangleq \\
\quad tar\,(path).attrib.perm.other.write \\
\textsf{pre}\ \ path \in \textsf{dom}\ tar \\
\quad ;
\end{array}
$$

2.3.6 checkReadPerm

Verifies if a process has read permission on a file

$checkReadPerm : Tar`Path \times Tar`Tar \times Tar`UserId \times Tar`GroupId \rightarrow \mathbb{B}$
$checkReadPerm\,(path, tar, uid, gid) \triangleq$
  $userReadPerm\,(path, tar, uid) \vee groupReadPerm\,(path, tar, gid) \vee otherReadPerm\,(path, tar)$
pre $path \in \mathsf{dom}\ tar$
  ;

$userReadPerm : Tar`Path \times Tar`Tar \times Tar`UserId \rightarrow \mathbb{B}$
$userReadPerm\,(path, tar, uid) \triangleq$
  if $(tar\,(path).attrib.user = uid)$
  then $tar\,(path).attrib.perm.user.read$
  else false
pre $path \in \mathsf{dom}\ tar$
  ;

$groupReadPerm : Tar`Path \times Tar`Tar \times Tar`GroupId \rightarrow \mathbb{B}$
$groupReadPerm\,(path, tar, gid) \triangleq$
  if $(tar\,(path).attrib.group = gid)$
  then $tar\,(path).attrib.perm.group.read$
  else false
pre $path \in \mathsf{dom}\ tar$
  ;

$otherReadPerm : Tar`Path \times Tar`Tar \rightarrow \mathbb{B}$
$otherReadPerm\,(path, tar) \triangleq$
  $tar\,(path).attrib.perm.other.read$
pre $path \in \mathsf{dom}\ tar$
  ;

2.3.7 checkRDOnly

Verifies if a process has read permission on a file

$checkRDOnly : Tar`Path \times Tar`Tar \times Tar`UserId \times Tar`GroupId \rightarrow \mathbb{B}$
$checkRDOnly\,(path, tar, -, -) \triangleq$
  let $perm = tar\,(path).attrib.perm$ in
  $\neg\,(perm.user.write \vee perm.group.write \vee perm.other.write)$
pre $path \in \mathsf{dom}\ tar$
  ;

2.3.8 checkFilePerm

Verifies if a process has permission to access a file

$checkFilePerm : System \times Tar`Path \times Flags\text{-}\mathsf{set} \rightarrow \mathbb{B}$
$checkFilePerm\,(sys, path, flags) \triangleq$
  let $proc = sys.process\,(sys.actualproc)$ in
  if $(\text{O\_RDONLY} \in flags)$
  then $checkReadPerm\,(path, sys.tarFileSystem, proc.userid, proc.groupid)$
  else if $(\text{O\_WRONLY} \in flags)$
    then $checkWritePerm\,(path, sys.tarFileSystem, proc.userid, proc.groupid)$
    else $checkReadPerm\,(path, sys.tarFileSystem, proc.userid, proc.groupid)$ $\wedge$
$checkWritePerm\,(path, sys.tarFileSystem, proc.userid, proc.groupid)$
pre $path \in \mathsf{dom}\ sys.tarFileSystem$
  ;

2.3.9 Setters and Getters

Gets the file contents associated with a file descritor.

$$
\begin{array}{l}
getFileContents : System \times \mathbb{N} \rightarrow FileContents \\
getFileContents\,(sys, filedes) \triangleq \\
\quad getFile\,(sys, filedes).contents \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \wedge \\
\quad\ getFile\,(sys, filedes).attrib.type = \mathrm{R} \\
\quad ;
\end{array}
$$

Gets the file associated with a file descriptor.

$$
\begin{array}{l}
getFile : System \times \mathbb{N} \rightarrow File \\
getFile\,(sys, filedes) \triangleq \\
\quad \textsf{let}\ path = sys.fds\,(filedes).file\ \textsf{in} \\
\quad sys.tarFileSystem\,(path) \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Gets the offset associated with a file descriptor.

$$
\begin{array}{l}
getOffset : System \times \mathbb{N} \rightarrow \mathbb{N} \\
getOffset\,(sys, filedes) \triangleq \\
\quad sys.fds\,(filedes).offset \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Gets the offset maximum associated with a file descriptor.

$$
\begin{array}{l}
getMaxOffset : System \times \mathbb{N} \rightarrow \mathbb{N} \\
getMaxOffset\,(sys, filedes) \triangleq \\
\quad sys.fds\,(filedes).maxoffset \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Gets the path associated with a file descriptor.

$$
\begin{array}{l}
getPath : System \times \mathbb{N} \rightarrow Path \\
getPath\,(sys, filedes) \triangleq \\
\quad sys.fds\,(filedes).file \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Sets system 'errno' variable.

$$
\begin{array}{l}
setErrno : System \times \textsf{char}^* \rightarrow System \\
setErrno\,(sys, err) \triangleq \\
\quad \mu\,(sys, errno \mapsto err);
\end{array}
$$

Sets a file descriptor 'st_atime' flag to false.

$$
\begin{array}{l}
setSt\text{-}ATime : System \times \mathbb{N} \rightarrow System \\
setSt\text{-}ATime\,(sys, filedes) \triangleq \\
\quad \textsf{let}\ \textsf{mk-}System\,(fs, procs, fds, aprocs, c, errno, us, en, mem, dev) = sys, \\
\qquad path = fds\,(filedes).file, \\
\qquad \textsf{mk-}File\,(attribs, conts) = fs\,(path), \\
\qquad \textsf{mk-}Attributes\,(type, perm, links, user, group, size, dates) = attribs, \\
\qquad \textsf{mk-}Dates\,(st\text{-}atime, st\text{-}ctime, st\text{-}mtime) = dates, \\
\qquad ndates = \textsf{mk-}Dates\,(\textsf{mk-}Date\,(st\text{-}atime.date, \textsf{false}), st\text{-}ctime, st\text{-}mtime), \\
\qquad nattribs = \textsf{mk-}Attributes\,(type, perm, links, user, group, size, ndates), \\
\qquad nfile = \textsf{mk-}File\,(nattribs, conts)\ \textsf{in} \\
\quad \textsf{mk-}System\,(fs \dagger \{path \mapsto nfile\}, procs, fds, aprocs, c, errno, us, en, mem, dev) \\
\textsf{pre}\ \ filedes \in \textsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Sets a file descriptor offset.

$$
\begin{array}{l}
setOffset : System \times \mathbb{N} \times \mathbb{N} \rightarrow System \\
setOffset\,(sys, filedes, newos) \triangleq \\
\quad \mathsf{let}\ \mathsf{mk}\text{-}System\,(fs, procs, fds, aprocs, c, errno, us, en, mem, dev) = sys, \\
\qquad \mathsf{mk}\text{-}OpenFileDescriptor\,(p, s, -, mo, f, md) = fds\,(filedes), \\
\qquad newfd = \mathsf{mk}\text{-}OpenFileDescriptor\,(p, s, newos, mo, f, md), \\
\qquad newfds = fds \dagger \{filedes \mapsto newfd\}\ \mathsf{in} \\
\quad \mathsf{mk}\text{-}System\,(fs, procs, newfds, aprocs, c, errno, us, en, mem, dev) \\
\mathsf{pre}\ \ filedes \in \mathsf{dom}\ sys.fds \\
\quad ;
\end{array}
$$

Puts data in system memory.

$$
\begin{array}{l}
setMemory : System \times \mathbb{N} \times Byte^* \rightarrow System \\
setMemory\,(sys, startd, data) \triangleq \\
\quad \mathsf{let}\ \mathsf{mk}\text{-}System\,(fs, procs, fds, aprocs, c, errno, us, en, mem, dev) = sys, \\
\qquad newmem = mem \dagger \{(startd + i - 1) \mapsto data\,(i)\ \mid \\
\qquad\qquad i \in \mathsf{inds}\ data\}\ \mathsf{in} \\
\quad \mathsf{mk}\text{-}System\,(fs, procs, fds, aprocs, c, errno, us, en, newmem, dev);
\end{array}
$$

2.3.10 applyMask

A bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared.

$$
\begin{array}{l}
applyMask : Tar`FilePermissions \times Tar`FilePermissions \rightarrow Tar`FilePermissions \\
applyMask\,(perm, umask) \triangleq \\
\quad \mathsf{let}\ user = \mathsf{mk}\text{-}Tar`Permissions\,(perm.user.read \wedge \neg umask.user.read, perm.user.write \wedge \\
\neg umask.user.write, perm.user.exec \wedge \neg umask.user.exec), \\
\qquad group = \mathsf{mk}\text{-}Tar`Permissions\,(perm.group.read \wedge \neg umask.group.read, perm.group.write \wedge \\
\neg umask.group.write, perm.group.exec \wedge \neg umask.group.exec), \\
\qquad other = \mathsf{mk}\text{-}Tar`Permissions\,(perm.other.read \wedge \neg umask.other.read, perm.other.write \wedge \\
\neg umask.other.write, perm.other.exec \wedge \neg umask.other.exec)\ \mathsf{in} \\
\quad \mathsf{mk}\text{-}Tar`FilePermissions\,(user, group, other)
\end{array}
$$

2.4 Objectifying

After we had resolved all our problems with a functional flavor, we use that functions to build an object. It's very simple to do this, we just have to an instance variable of type System and apply our functions to it with the correct arguments.

```
end Tar
class TarObj is subclass of Tar
instance variables
 system : System := init ();


Mkdir : Path × Mode-t-set →o ℤ
Mkdir (path, mode) △
   def r = mkdir (path, mode, system) in
   (     system := r.#2;
         return r.#1
   );

Open : Path × Flags-set × Mode-t-set →o ℤ
Open (path, flags, mode) △
   def r = open (path, flags, mode, system) in
   (     system := r.#2;
         return r.#1
   );

Write : ℕ × ℕ × ℕ →o ℤ
Write (fildes, buf, nbyte) △
   def r = write (system, fildes, buf, nbyte) in
   (     system := r.#2;
         return r.#1
   );

Read : ℕ × ℕ × ℕ →o ℤ
Read (fildes, buf, nbyte) △
   def r = read (system, fildes, buf, nbyte) in
   (     system := r.#2;
         return r.#1
   )
end TarObj
```

2.5 Testing

Our model was not extensively tested, not only because of lack of time but also because it was a bit dificult to see the outcome when using a realistic size file system.

Anyway, as a tool to help in testing, we developed a *Haskell* script that scans a directory and recursively its sub-directories to produce a *Tar* VDM type. We recommend its use in future developments of this project, as it is very easy to use: just run it, choose which directory to scan and then choose a file for the output to be written.

3 Conclusion

With this work we hope to have set the ground for a VDM specification of a POSIX-like file system. We have implemented a data type and some basic operations to suport the file system *modus operandi*. For this purpose, we followed a list of requirements for each function developed by the *Open Linux Verification*. It has not been possible to model all the requirements because (a) in some cases because it wouldn't be possible to do it with VDM itself; (b) in other cases, lack of time led us to decide not to make the model too complex. Overcoming these limitations is work for the future.

Concerning basic data type definitions, we first thought in some invariants that would prevent some errors from happening, for instance preventing that a link points to a file that doesn't exist, or that only files that exist are associated with a file descriptor. These invariants would turn the file system safer but they weren't mentioned in any specification, so we didn't introduce them because we didn't want to "narrow" the problem.

In modelling the functionality we had to take decisions wherever requirements weren't clear. Some other times there were requirements that overlapped others. These situations are properly

mentioned in the report.

One of the great dificulties of this work was to choose a standard requirements document to follow. This happened because of some involved aspects that weren't possible to implement, or of requirements that were missing, or even because almost all of them had parts that weren't very clear.

Although there are lots of aspects to improve, considering the dimension of the project and the time avaiable, we think that in general the outcome is positive. This has helped us in seeing how to approach big sized problems from a formal point of view.

The effort put into this assignment was roughly 100 hours of work.

References

[1] Fitzgerald, J., and Larsen, P. G. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1st edition, 1998.

[2] Fitzgerald, J., Larsen, P. G., Mukherjee, P., Plat, N., and Verhoef M. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.

[3] Joshi R., and Holzmann, G. J. *A Mini Challenge: Build a Verifiable Filesystem*. California Institute of Technology.

[4] `http://en.wikipedia.org`

[5] `http://linuxtesting.org/`

[6] `http://opengroup.org/`

[7] `http://openss7.org/`

[8] `http://uw714doc.sco.com/`

[9] `http://www.vdmtools.jp/en/`