2.5. Case study: Unix® file system layering and naming

Unix is a family of operating systems that trace their lineage back to the Unix operating system that was developed by Bell Telephone Laboratories for the Digital Equipment Corporation PDP line of minicomputers in the late 1960s and early 1970s [Suggestions for Further Reading 2.2], and before that to the Multics[*] operating system in the early 1960s [Suggestions for Further Reading 1.7.5 and 3.1.4]. Today there are many flavors of Unix with complex historical relationships; a few examples include GNU/Linux, versions of GNU/Linux distributed by different organizations (e.g., Redhat, Ubuntu), Darwin (a Unix operating system that is part of Apple's operating system Mac OS X), and several flavors of BSD Unix. Some of these are directly derived from the early Unix operating system; others provide similar interfaces but have been implemented from scratch. Some are the result of an effort by a small group of programmers and others are the result of an effort by many. In the latter case, it is even unclear how to exactly name the operating system because substantial parts come from different teams[†]. The collective result of all these efforts is that operating systems of the Unix family run on a wide range of computers, including personal computers, server computers, parallel computers, and embedded computers. Most of the Unix interface is an official standard[‡] and non-Unix operating systems often support this standard too. Because the source code of some versions is available to the public, one can easily study Unix.

This case study examines the various ways in which Unix uses names in its design. In the course of examining how Unix implements its naming scheme, we will also incidentally get a first-level overview of how the Unix file system is organized.

2.5.1. Application programming interface for the Unix file system

A program can create a file with a user-chosen name, read and write the file's content, and set and get a file's *metadata*. Example metadata include the time of last modification, the user ID of the file's owner, and access permissions for other users. (For a full discussion of metadata see section 3.1.2.) To organize their files, users can group them in directories with user-chosen names, creating a *naming network*. Users can also graft a naming network stored on a storage device onto an existing naming network, allowing naming networks for different

[*] The name Unix evolved from Unics, which was a word joke on Multics.

[†] We use "Linux" for the Linux kernel while we use "GNU/Linux" for the complete system, recognizing that his naming convention is not perfect either, because there are pieces of the system that are neither GNU software or part of the kernel (e.g., the X Window System, see sidebar 4.4).

[‡] POSIX® (Portable Operating System Interface), Federal Information Processing Standards (FIPS) 151-2. FIPS 151-2 adopts ISO/IEC 9945-1: 2003 (IEEE Std. 1003.1: 2001) Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface (API) [C Language].

Table 2.1: Unix file system application programming interface

Procedure	Brief description
OPEN (*name*, *flags*, *mode*)	Open file *name*. If the file doesn't exist and *flags* is set, create file with permissions *mode*. Set the file cursor to 0. Returns a file descriptor.
READ (*fd*, *buf*, *n*)	Read *n* bytes from the file at the current cursor and increase the cursor by the number of bytes read.
WRITE (*fd*, *buf*, *n*)	Write *n* bytes at the current cursor and increase the cursor by the bytes written.
SEEK (*fd*, *offset*, *whence*)	Set the cursor to *offset* bytes from beginning, end, or current position.
CLOSE (*fd*)	Delete file descriptor. If this is the last reference to the file, delete the file.
FSYNC (*fd*)	Make all changes to the file durable
STAT (*name*)	Read metadata of file.
CHMOD, CHOWN, etc.	Various procedures to set specific metadata.
RENAME (*from_name*, *to_name*)	Change name from *from_name* to *to_name*
LINK (*name*, *link_name*)	Create a hard link *link_name* to the file *name*.
UNLINK (*name*)	Remove *name* from its directory. If *name* is the last name for a file, remove file.
SYMLINK (*name*, *link_name*)	Create a symbolic name *link_name* for the file *name*.
MKDIR (*name*)	Create a new directory named *name.*
CHDIR (*name*)	Change current working directory to *name.*
CHROOT (*name*)	Change the default root directory to *name.*
MOUNT (*name*, *device*)	Graft the file system on *device* onto the name space at *name.*
UNMOUNT (*name*)	Unmount the file system at *name.*

devices to be incorporated into a single large naming network. To support these operations, Unix provides the application programming interface (API) shown in table 2.1.

To tackle the problem of implementing this API, Unix employs a divide-and-conquer strategy. The Unix file system makes use of several hidden layers of machine-oriented names

Table 2–2: The naming layers of Unix.

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
Path name layer	Organize files into naming hierarchies.	↓
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑
File layer	Organize blocks into files.	machine-oriented names
Block layer	Identify disk blocks.	↓

(that is, addresses), one on top of another, to implement files. It then applies the Unix durable object naming scheme to map user-friendly names to these files. Table 2–2 illustrates this structure.

In the rest of this section we work our way up from the bottom layer of table 2–2 to the top layer, proceeding from the lowest layer of the system up toward the user. This description corresponds closely to the implementation of Version 6 Unix, one of the earlier versions of Unix, which dates back to the early 1970s. Version 6 is well documented [Suggestions for Further Reading 2.2.2] and captures the important ideas that are found in any modern Unix file systems, but modern versions are more complex; they provide better robustness, and handle large files, many files, etc. more efficiently. In a few places we will point out some of these differences, but the reader is encouraged to consult papers in the file system literature to find out how modern Unix file systems work.

2.5.2. *The block layer*

At the bottom layer Unix names some physical device such as a magnetic disk, floppy disk, or magnetic tape that can store data durably. The storage on such a device is divided into fixed-size units, called *blocks* in Unix. For a magnetic disk (see sidebar 2.2), a block corresponds to a small number of disk sectors. A block is the smallest allocation unit of disk space, and its size is a trade-off between several goals. A small block reduces the amount of disk wasted for small files; if many files are smaller than 4 kilobytes, a 16 kilobyte block size wastes space. On the other hand, a very small block size may incur large data structures to keep track of free and allocated blocks. In addition, there are performance considerations that

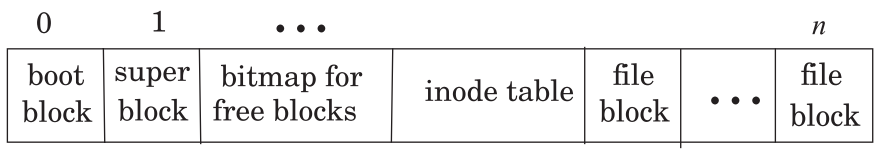


Figure 2.19: Possible disk layout for a simple file system

impact the block size, some of which we discuss in chapter 6. Version 6 Unix used 512 byte blocks, but modern Unix file systems often use 8 kilobyte blocks.

The names of these blocks are numbers, which typically correspond to the offset of the block from the beginning of the device. In the bottom naming layer, a storage device can be viewed as a context that binds block numbers to physical blocks. The name-mapping algorithm for a block device is simple: it takes as input a block number and returns the block. Actually, we don't really want the block itself—that would be a pile of iron oxide—what we want is the *contents* of the block, so the algorithm actually implements a fixed mapping between block name and block contents. If we represent the storage device as a linear array of blocks, then the following code fragment implements the name-mapping algorithm:

```
procedure BLOCK_NUMBER_TO_BLOCK (integer b) returns block instance
    return device[b]
```

In this algorithm the variable name *device* refers to some particular physical device. In many devices the mapping is more complicated. For example, a hard drive might keep a set of spare blocks at the end, and rebind the block numbers of any blocks that go bad to spares. The hard drive may itself be implemented in layers, as will be seen in section 8.5.4.

Name discovery: The names of blocks are integers from a compact set, but the block layer must keep track of which blocks are in use and which are available for assignment. As we will see, the file system in general has a need for a description of the layout of the file system on disk. As an anchor for this information, the Unix file system starts with a *super block*, which has a well-known name (e.g., 1). The super block contains, for example, the size of the file system's disk in blocks. (Block 0 typically stores a small program that starts the operating system; see sidebar 5.3.)

Different implementations of Unix use different representations for the list of free blocks. Version 6 Unix keeps a list of block numbers of unused blocks in a linked list that is stored in some of the unused blocks. The block number of the first block of this list is stored in the super block. A call to allocate a block leads to a procedure in the block layer that searches the list array for a free block, removes it from the list, and returns that block's block number.

Modern Unix file systems often use a bitmap for keeping track of free blocks. Bit *i* in the bitmap records whether block *i* is free or allocated. The bitmap itself is stored at a well-known location on the disk (e.g., right after the super block). Figure 2.19 shows a possible disk layout for a simple file system. It starts with the super block, followed by a bitmap that records which disk blocks are in use. After the bitmap comes the inode table, which has one entry for each file (as explained next), followed by blocks that are either free or allocated to some file. The super block contains the size of the bitmap and inodes table in blocks.

2.5.3. The file layer

Users need to store items that are larger than one block in size, and that may grow or shrink over time. To support such items Unix introduces a next naming layer for *files*. A file is a linear array of bytes of arbitrary length. The file system needs to record in some way which blocks belong to each file. To support this requirement, Unix creates an index node, or *inode* for short, as a container for metadata about the file. Our initial declaration of an inode is:

```
structure inode
    integer block_numbers[N]    // the numbers of the blocks that constitute the file
    integer size                // the size of the file in bytes
```

The inode for a file is thus a context in which the various blocks of the file are named by integer block numbers. With this structure, a simplified name-mapping algorithm for resolving the name of a block in a file is as follows:

```
procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]
```

Version 6 Unix uses a more sophisticated algorithm for mapping the *index*-th block of an inode to a block number. It reserves some of the blocks of the inode array for creating files that are larger than *N* blocks. These special blocks do not contain data, but more block numbers, so they are called *indirect blocks*. For example, with a block size of 512 bytes and an *index* of 2 bytes (as in Version 6), an indirect block can contain 256 2-byte block numbers. For small files, only the last block is an indirect block, and then the maximum file size is *(N – 1)* + 256 blocks. In Version 6 Unix *N* is 8, and the maximum file size for small files is 131 kilobytes. Problem set *1* explores some design trade-offs to allow the file system to support large files.

To support larger files, Version 6 uses a different representation: the first 7 entries in *i.block_numbers* are singly-indirect blocks and the last one is a doubly-indirect block (blocks that contain block numbers of indirect blocks). This choice allows for 65536 blocks or 32 megabytes*. Some modern Unix file system use different representations or more sophisticated data structures, such as B+ trees, to implement files.

* The implementation of Version 6, however, restricts the maximum number of blocks per file to 2^{15}.

Unix allows users to name any particular byte in a file by layering the previous two naming schemes and specifying the byte number as an offset from the beginning of the file:

```
procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block instance
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

Version 6 used for *offset* a 3-byte number, which limits the maximum file size to 2^{24} bytes. Modern Unix file systems use a 64-bit number. The procedure returns the entire block that contains the named byte. As we will see in section 2.5.11, READ uses this procedure to return the requested bytes.

2.5.4. The inode number layer

Instead of passing inodes themselves around, it would be more convenient to name them and pass their names around. To support this feature, Unix provides another naming layer that names inodes by an inode number. A convenient way to implement this naming layer is to employ a table that directly contains all inodes, indexed by inode number. Here is the naming algorithm:

```
procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode instance
    return inode_table[inode_number]
```

where *inode_table* is an object that is stored at a fixed location on the storage device (e.g., at the beginning). The name-mapping algorithm for *inode_table* just returns the starting block number of the table.

Name discovery: inode numbers, like disk block numbers, are a compact set of integers, and again the inode number layer must keep track of which inode numbers are in use and which are free to be assigned. As with block number assignment, different Unix implementations use various representations for a list of free inodes and provide calls to allocate and deallocate inodes. In the simplest implementation, the inode contains a field recording whether it is free or not.

By putting these three layers together, we obtain the following procedure:

```
procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
                                                            returns block instance
    inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

This procedure resolves an inode number and an offset to the block that contains the byte at that offset. This procedure traverses 3 layers of naming. There are numbers for storage blocks, numbered indexes for blocks belonging to an inode, and numbers for inodes.

2.5.5. The file name layer

Numbers are convenient names for use by a computer (numbers can be stored in fixed-length fields that simplify storage allocation) but are inconvenient names for use by people (numbers have little mnemonic value). In addition, block and inode numbers specify a location, so if it becomes necessary to rearrange the physical storage, the numbers must change, which is again inconvenient for people. Unix deals with this problem by inserting a naming layer whose sole purpose is to hide the metadata of file management. Above this layer is a user-friendly naming scheme for durable objects—files and input/output devices. This naming scheme again has several layers. The most visible component of the durable object naming scheme is the *directory*. In Unix, a directory is a context containing a set of bindings between character-string names and inode numbers.

To create a file, Unix allocates an inode, initializes its metadata, and binds the proposed name to that inode in some directory; as the file is written the file system allocates blocks to the inode.

By default Unix adds the file to the current working directory. The current working directory is a context reference to the directory in which the active application is working. The form of the context reference is just another inode number. If *wd* is the name of the state variable that contains the working directory for a running program (called a *process* in Unix), one can look up the inode number of the just-created file by supplying *wd* as the second argument to a procedure such as:

```
procedure NAME_TO_INODE_NUMBER (character string filename, integer dir) returns integer
    return LOOKUP (filename, dir)
```

The procedure CHDIR, whose implementation we describe later, allows a process to set *wd*.

To represent a directory, Unix reuses the mechanisms developed so far: it represents directories as files. By convention a file that represents a directory contains a table that maps file names to inode numbers. For example, figure 2.21 is a directory with two file names (program and paper), which are mapped to inode numbers 10 and 12, respectively. In Unix Version 6, the maximum length of a name is 14 bytes and the entries in the table have a fixed length of 16 bytes (14 for the name and 2 for the inode number). Modern Unix file systems allow for variable-length names, and the table representation is more sophisticated.

File name	Inode Number
program	10
paper	12

Figure 2.20: A directory

To record whether an inode is for a directory or a file, Unix extends the inode with a type field:

```
structure inode
    integer block_numbers[N]    // the numbers of the blocks that constitute the file
    integer size                // the size of the file in bytes
    integer type                // type of file: regular file, directory,…
```

MKDIR creates a zero-length file (directory) and sets *type* to DIRECTORY. Extensions introduced later will add additional values for *type*.

With this representation of directories and inodes, LOOKUP is as follows:

```
1   procedure LOOKUP (character string filename, integer dir) returns integer
2       block instance b
3       inode instance i ← INODE_NUMBER_TO_INODE (dir)
4       if i.type ≠ DIRECTORY then return FAILURE
5       for offset from 0 to i.size – 1 do
6           b ← INODE_NUMBER_TO_BLOCK (offset, dir)
7           if STRING_MATCH (filename, b) then
8               return INODE_NUMBER (filename, b)// return inode number for filename
9           offset ← offset + BLOCKSIZE          // increase offset by block size
10      return FAILURE
```

LOOKUP reads the blocks that contain the data for the directory *dir* and searches for the string *filename* in the directory's data. It computes the block number for the first block of the directory (line *6*) and the procedure STRING_MATCH (no code shown) searches that block for an entry for the name *filename*. If there is an entry, INODE_NUMBER (no code shown) returns the inode number in the entry (line *8*). If there is no entry, LOOKUP computes the block number for the second block, and so on, until all blocks of the directory have been searched. If none of the blocks contain an entry for *filename*, LOOKUP returns an error (line *10*). As an example, an invocation of LOOKUP ("program", *dir*), where *dir* is the inode number for the directory of figure 2.21, would return the inode number 10.

2.5.6. The path name layer

Having all files in a single directory makes it hard for users to keep track of large numbers of files. Enumerating the contents of a large directory would generate a long list that is organized simply (e.g., alphabetically) at best. To allow arbitrary groupings of user files, Unix permits users to create named directories.

A directory can be named just like a file, but the user also needs a way of naming the files in that directory. The solution is to add some structure to file names: for example, "projects/paper", in which "projects" names a directory and "paper" names a file in that directory. Structured names such as these are examples of path names. Unix uses a virgule (forward slash) as a separator of the components of a path name; other systems choose different separator characters such as period, back slash, or colon. With these tools, users can create a hierarchy of directories and files.

The name-resolving algorithm for path names can be implemented by layering a recursive procedure over the previous directory lookup procedure:

```
1  procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
2      if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
3      else
4          dir ← LOOKUP (FIRST (path), dir)
5          path ← REST (path)
6          return PATH_TO_INODE_NUMBER (path, dir)
```

The function PLAIN_NAME (*path*) scans its argument for the Unix standard path name separator (forward slash) and returns TRUE if it does not find one. If there is no separator, the program resolves the simple name to an inode number in the requested directory (line *2*). If there is a separator in *path*, the program takes it to be a path name and goes to work on it (lines *4* through *6*). The function FIRST peels off the first component name from the path and REST returns the remainder of the path name. Thus, for example, the call PATH_TO_NAME (“projects/paper”, *wd*) results in the recursive call PATH_TO_NAME (“paper”, *dir*), where *dir* is the inode number for the directory “projects”.

With path names, one often has to type names with many components. To address this annoyance, Unix supports a change directory procedure, CHDIR, allowing a process to set their working directory:

```
procedure CHDIR (path character string)
    wd ← PATH_TO_INODE_NUMBER (path, wd)
```

When a process starts, it inherits the working directory from the parent process that created this process.

2.5.7. *Links*

To refer to files in directories other than the current working directory still requires typing long names. For example, while we are working in the directory “projects” —after calling CHDIR (“projects”)—we might have to refer often to the file “Mail/inbox/new-assignment”. To address this annoyance, Unix supports synonyms known as *links*. In the example, we might want to create a link for this file in the current working directory, “projects”. The LINK procedure

```
LINK (“Mail/inbox/new-assignment”, “assignment”)
```

makes “assignment” a synonym for “Mail/inbox/new-assignment” in “projects”, if “assignment” doesn’t exist yet. (If it does, LINK will return an error saying “assignment” already exists.) With links, the directory hierarchy turns from a strict hierarchy into a directed graph. (Unix allows links only to files, not to directories, so the graph is not only directed, it is acyclic. We shall see why in a moment.)

Unix implements links simply as bindings in different contexts that map different file names to the same inode number, so links don’t require any extension to the naming scheme developed so far. For example, if the inode number for “new-assignment” is 481, then the directory “Mail/inbox” contains an entry {“new-assignment”, 481} and after the above command is executed the directory “projects” contains an entry {“assignment”, 481}. In Unix jargon, “projects/assignment” is now linked to “Mail/inbox/new-assignment”.

When a file is no longer needed, a process can remove a file using UNLINK (*filename*), indicating to the file system that the name *filename* is no longer in use. UNLINK removes the binding of *filename* to its inode number from the directory that contains *filename*. The file system also puts *filename*’s inode and the blocks of *filename*’s inode on the free list if this binding is the last one containing the inode’s number.

Before we added links, a file was bound to a name in only one directory, so if a process asks to delete the name from that directory the file system can also delete the file. But now that links have been added, when a process asks to delete a name there may still be names in other directories bound to the file, in which case the file shouldn’t be deleted. This raises the question when should a file be deleted? Unix deletes a file when a process removes the last binding for a file. Unix implements this policy by keeping a reference count in the inode:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

Whenever it makes a binding to an inode, the file system increases the reference count of that inode. To delete a file, Unix provides an UNLINK(*filename*) entry, which deletes the binding specified by *filename*. The file system at the same time decreases the reference count in the corresponding inode by one. If the decrease causes the reference count to go to zero, that means there are no more bindings to this inode, so Unix can free the inode and its corresponding blocks. For example, UNLINK (“Mail/inbox/new-assignment”) removes the directory entry “new-assignment” in the directory “Mail/inbox”, but not “assignment”, because after the unlink the *refcnt* in inode 481 will be 1. Only after calling UNLINK (“assignment”) will the inode 481 and its blocks be freed.

Using reference counts works only if there are no cycles in the naming graph. To ensure that the Unix naming network is a directed graph without cycles, Unix forbids links to directories. To see why cycles are avoided, consider a directory “a”, which contains a directory “b”. If a user types link (“a/b/c”, “a”) in the directory that contains “a”, then Unix would return an error and not perform the operation. If Unix had performed this operation, it would have created a cycle from “c” to “a”, and would have increased the reference count in the inode of “a” by one. If a user then typed unlink (“a”), the name “a” is removed, but the inode and the blocks of “a” wouldn’t be removed, because the reference count in the inode of “a” is still positive (because of the link from “c” to “a”). But once the name “a” would be removed, the user would not be able to name the directory “a” any more and wouldn’t be able to remove it either. In that case, the directory “a” and its subdirectories would be disconnected from the naming graph, but Unix would not remove it because the reference count in the inode of “a” is still positive. It is possible to detect this situation, for example by using garbage collection, but it is expensive to do so. Instead, the designers chose a simpler solution: don’t allow links to directories, which rules out the possibility of cycles.

There are two special cases, however. First, by default each directory contains a link to itself; Unix reserves the string "." (a single dot) for this purpose. The name "." thus allows a process to name the current directory without knowing which the directory it is. When a directory is created, the directory's inode has a reference count of two: one for the inode of the directory and one for the link ".", because it points to itself. Because "." introduces a cycle of length 0, there is no risk that part of the naming network will become disconnected when removing a directory. When unlinking a directory, Unix just decreases the reference count of the directory's inode by 2.

Second, by default each directory also contains a link to a parent directory; Unix reserves the string ".." (two consecutive dots) for this purpose. The name ".." allows a process to name a parent directory and, for example, move up the file hierarchy by invoking CHDIR (".."). The link doesn't create problems. Only when a directory has no other entries than "." and ".." can it be removed. If a user wants to remove a directory "a", which contains a directory "b", then Unix refuses to do so until the user first has removed "b". This rule ensures that the naming network cannot become disconnected.

2.5.8. Renaming

Using LINK and UNLINK, Version 6 implemented RENAME (*from_name*, *to_name*) as follows:

```
1    UNLINK (to_name)
2    LINK (from_name, to_name)
3    UNLINK (from_name)
```

This implementation, however, has an undesirable property. RENAME is often used by programs to change a working copy of a file into the official version; for example, a user may be editing a file "x". The text editor actually makes all changes to a temporary file "#x". When the user saves the file, the editor renames the temporary file "#x" to "x".

The problem with implementing RENAME using LINK and UNLINK is that if the computer fails between steps 1 and 2 and then restarts, the name *to_name* ("x" in this case) will be lost, which is likely to surprise the user; the user is unlikely to know that the file still exists but under the name "#x". What is really needed is that "#x" be renamed to "x" in a single, atomic operation, but that requires atomic actions, which are the topic of chapter 9.

But, without atomic actions, it is possible to implement the following slightly weaker specification for RENAME: if *to_name* already exists, an instance of *to_name* will always exist, even if the system should fail in the middle of RENAME. This specification is good enough for the editor to do the right thing and is what modern versions of Unix provide.

Modern versions of Unix implement this specification in essence as follows:

```
1    LINK (from_name, to_name)
2    UNLINK (from_name)
```

But, because one cannot link to a name that already exists, RENAME implements the effects of these two calls by manipulating the file system structures directly. RENAME first changes the inode number in the directory entry for *to_name* to the inode number for *from_name* on disk.

Then, RENAME removes the directory entry for *from_name*. If the file system fails between these two steps, then on recovery the file system must increase the reference count in *from_name*'s inode because both *from_name* and *to_name* are pointing to the inode. This implementation ensures that if *to_name* exists before the call to RENAME, it will continue to exist, even if the computer fails during RENAME.

2.5.9. The absolute path name layer

Unix provides each user with a personal directory, called a user's *home directory*. When a user logs on to a Unix system, Unix will start a command interpreter (known as the *shell*) through which a user can interact with the system interactively. The shell starts with the working directory (*wd*) set to the inode number of the user's home directory. With the above procedures, users can create personal directory trees to organize the files in their home directory.

But having several personal directory trees does not allow one user to share files with another. To do that, one user needs a way of referring to the names of files that belong to another user. The easiest way to accomplish that is to bind a name for each user to that user's top-level directory, in some context that is available to every user. But then there is a requirement to name this system-wide context. There are typically needs for other system-wide contexts, such as a directory containing shared program libraries. To address these needs with a minimum of additional mechanism, Unix provides a universal context, known as the *root* directory. The root directory contains bindings for the directory of users, the directory containing program libraries, and any other widely-shared directories. The result is that all files of the system are integrated into a single directory tree (with restricted cross-links) based on the root.

This design leaves a name discovery question: how can a user name the root directory? Recall that name lookup requires a context reference—the name of a directory inode—and until now that directory inode has been supplied by the working directory state variable. To implement the root, Unix simply declares inode number 1 to be the inode for the root directory. This *well-known name* can then be used by any user as the starting context in which to look up the name of a shared context, or another user (or even to look up one's own name, to set the working directory when logging in).

Unix actually provides two ways to refer to things in the root directory. Starting from any directory in the system, one can utter the name ".." to name that directory's parent, "../.." to name the directory above that, and so on until the root directory is reached. A user can tell that the root directory is reached, because ".." in the root directory names the root directory. That is, in the root directory, both "." and ".." are links to the root directory. The other way is with absolute path names, which in Unix are names that start with a "/", for example "/Alice/Mail/inbox/new-assignment".

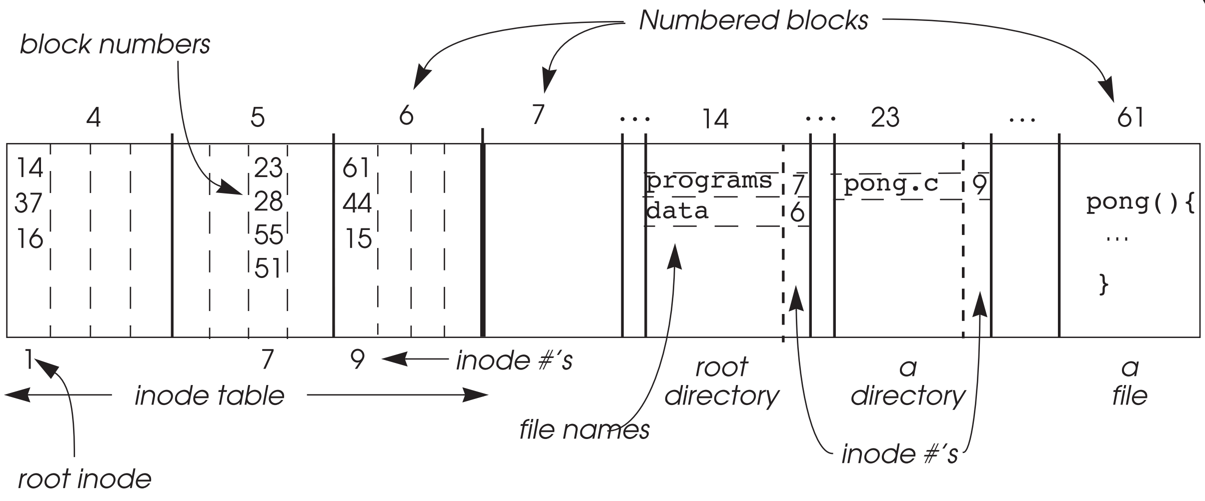


Figure 2.22: Example disk layout for a Unix file system, refining figure 2.19 by focusing in on the inode table and data blocks. The inode table is a group of contiguous blocks starting at a well-known address, found in the super block (not shown). In this example, blocks 4, 5, and 6 contain the inode table, while blocks 7–61 contain directories and files. The root inode is by convention the well-known inode #1. Typically, inodes are smaller than a block, so in this example there are four inodes in each block. Blocks #14, #37, and #16 constitute the root directory, while block #23 is the first of 4 blocks of the directory named "/programs", and block #61 is the first block of the 3-block file "/programs/pong.c".

To support absolute path names as well as relative path names we need one more layer in the naming scheme:

```
1   procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
2       if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
3       else return PATH_TO_INODE_NUMBER(path, wd)
```

At this point we have completed a naming scheme that allows us to name and share durable storage on a single disk. For example, to find the blocks corresponding to the file "/programs/pong.c" with the information in figure 2.22, we start by finding the inode table, which starts at a block number (block 4 in our example) stored in the super block (not shown in this figure, but see figure 2.19). From there we locate the root inode (which is known to be inode number 1). The root inode contains the block numbers that in turn contain the blocks of the root directory; in the figure the root starts in block number 14. Block 14 lists the entries in the root directory: "programs" is named by inode number 7. The inode table says that data for inode number 7 starts in block number 23, which contains the contents of the *programs* directory. The file "pong.c" is named by inode number 9. Referring once more to the inode table, to see where inode 9 is stored, the data corresponding to inode 9 starts in block number 61. In short, directories and files are carefully laid out so that all information can be found by starting from the well-known location of the root inode.

The default root directory in Version 6 is inode 1. Since Version 7, Unix also provides a call, CHROOT, to change the root directory for a process. For example, Unix may run a Web server in the corner of the Unix name space by changing its root directory to, for example, "/tmp". After this call, the root directory for the Web server corresponds to the inode number of the directory "/tmp" and ".." in "/tmp" is a link to "/tmp". Thus, the server can name only directories and files below "/tmp".

2.5.10. The symbolic link layer

To allow users to name files on other disks, Unix supports an operation to attach new disks to the name space. A user can choose the name under which each device is attached: for example, the procedure

MOUNT ("/dev/fd1", "/floppy")

grafts the directory tree stored on the physical device named "/dev/fd1" onto the directory "/floppy". (This command demonstrates that each device also has a name in the same object name space we have been describing; the file corresponding to a device typically contains information about the device itself.) Typically mounts do not survive a shutdown: after a reboot the user has to explicitly remount the devices. It is interesting to contrast the elegant Unix approach with the DOS approach, in which devices are named by fixed one-character names (e.g., "C:").

The Unix file system implements MOUNT by recording in the in-memory inode for "floppy" that a file system has been mounted on it and keeps this inode in memory until at least the corresponding UNMOUNT. In memory, Unix also records the device and the root inode number of the file system that has been mounted on it. In addition, Unix records in the in-memory version of the inode for "/dev/fd1" what its parent inode is.

The information for mount points is all recorded in volatile memory instead of on disk, and doesn't survive a computer failure. After a failure, the system administrator or a program must invoke MOUNT again. Supporting MOUNT also requires a change to the file name layer: if LOOKUP runs into an inode on which a file system is mounted, it uses the root inode of the mounted file system for the lookup.

UNMOUNT undoes the mount.

With mounted file systems, synonyms become a harder problem, because per mounted file system there is an address space of inode numbers. Every inode number has a default context: the disk on which it is located. Thus there is no way for a directory entry on one disk to bind to an inode number on a different disk. There are several ways to approach this problem, two of which are: (1) make inodes unique across all disks or (2) create synonyms for files on other disks in a different way. Unix chooses the second approach, by using indirect names called *symbolic* or *soft links*, which bind a file name to another file name. Most systems use method (2) because of the complications that would be involved in trying to keep inode numbers universally unique, small in size, and fast to resolve.

Using the procedure SYMLINK, users can create synonyms for files in the same file system or for files in mounted file systems. Unix implements the procedure SYMLINK by allowing the type field of an inode to be a SYMLINK, which tells whether the blocks associated with the inode contain data or a path name:

```
structure inode
    integer block_numbers[N]
    integer size
    integer type        // Type of inode: regular file, directory, symbolic link,…
    integer refcnt
```

If the *type* field has value SYMLINK, then the data in the array *blocks*[*i*] actually contains the characters of a path name rather than a set of inode numbers.

Soft links can be implemented by layering them over GENERALPATH_TO_NODE_NUMBER:

```
procedure PATHNAME_TO_INODE (character string filename) returns inode instance
    inode instance i
    inode_number ← GENERALPATH_TO_INODE_NUMBER (filename)
    i ← INODE_NUMBER_TO_INODE (inode_number)
    if i.type = SYMBOLIC then
        i = GENERALPATH_TO_INODE_NUMBER (COERCE_TO_STRING (i.block_numbers))
    return i
```

We now have two types of synonyms. In Unix, a direct binding to an inode number is called a *hard link*, to distinguish it from a soft link. Continuing an earlier example, a soft link to “Mail/inbox/new-assignment” would contain the string “Mail/inbox/new-assignment”, rather than the inode number 481. A soft link is an example of an *indirect name*: it binds a name to another name in the same name space, while a hard link binds a name to an inode number, which is a name in a lower-layer name space. As a result, the soft link depends on the file name “Mail/inbox/new-assignment”; if the user changes the file’s name or deletes the file, then “projects/assignment”, the link, will end up as a dangling reference (Section 3.1.6 discusses dangling references). But because it links by name rather than by inode number a soft link can point to a file on a different disk.

Recall that Unix forbids cycles of hard links, so that it can use reference counts to detect when it is safe to reclaim the disk space for a file. However, Unix lets you form cycles with soft links: a name deep down in the tree can, for example, name a directory high up in the tree. The resulting structure is not a directed acyclic graph any more, but a fully general naming network. Using soft links, a program can even invoke SYMLINK (“cycle”, “cycle”), creating a synonym for a file name that doesn’t have a file associated with it! If a process opens such a file, it will follow the link chain only a certain number of steps before reporting an error such as “Too many levels of soft links”.

Soft links have another interesting behavior. Suppose the working directory is “/Scholarly/programs/www”, and that this working directory contains a symbolic link named “CSE499-web” to “/Scholarly/CSE499/www”. The following calls:

```
CHDIR (“CSE499-web”)
CHDIR (“..”)
```

leaves the caller in “/Scholarly/CSE499” rather than back where the user started. The reason is that “..” is resolved in the new default context, “/Scholarly/CSE499/www”, rather than what

Table 2–3: The naming layers of Unix, with details of the naming scheme of each layer.

Layer	Names	Values	Context	Name-mapping algorithm	
Symbolic link	Path names	Path names	The directory hierarchy	PATHNAME_TO_GENERAL_PATH	↑
Absolute path name	Absolute path names	Inode numbers	The root directory	GENERALPATH_TO_INODE_NUMBER	user-oriented names
Path name	Relative path names	Inode numbers	The working directory	PATH_TO_INODE_NUMBER	↓
File name	File names	Inode numbers	A directory	NAME_TO_INODE_NUMBER	machine-user interface
Inode number	Inode numbers	Inodes	The inode table	INODE_NUMBER_TO_INODE	↑
File	Index numbers	Block numbers	An inode	INDEX_TO_BLOCK_NUMBER	machine-oriented names
Block	Block numbers	Blocks	The disk drive	BLOCK_NUMBER_TO_BLOCK	↓

might have been the intended context, "/Scholarly/programs/www". This behavior may be desirable or not, but it is a direct consequence of the naming semantics chosen for Unix; the Plan 9 system has a different plan*, which is also explored in exercises *Ex. 3.2* and *Ex. 3.3*.

In summary, much of the power of the Unix object naming scheme comes from its layers of naming. Table 2–3 reprises table 2–2, this time showing the name, value, context, and pseudocode procedure used at each layer interface. (Although we have examined each of the layers in this table, the algorithms we have demonstrated have in some cases bridged across layers in ways not suggested by the table.) The general design technique has been to introduce for each problem another layer of naming, an application of the principle *decouple modules with indirection*.

2.5.11. Implementing the file system API

In the process of describing how the Unix file system is structured, we saw how Unix implements CHDIR, MKDIR, LINK, UNLINK, RENAME, SYMLINK, MOUNT, and UNMOUNT. We complete the description of the file system API by describing the implementation of OPEN, READ, WRITE, and CLOSE. Before describing their implementation, we describe what features they must support.

The file system allows users to control who has access to their files. An owner of a file can specify with what permissions other users can make accesses to the file. For example, the

* Rob Pike. Lexical File Names in Plan 9 or Getting Dot-Dot Right. *Proceedings of the 2000 USENIX Technical Conference* (2000), San Diego, pages 85–92.

owner may specify that other users have permission only to read a file, but not to write it. OPEN must check whether the caller has the appropriate permissions. As a sophistication, Unix allows a file to be owned by a group of users. Chapter 11 discusses security in detail, so we will skip the details here.

The file system records time stamps that capture the date and time of the last access, last modification to a file, and the last change to a file's inode. This information is important for programs such as incremental backup, which must determine which files have changed since the last time backup ran. The file system procedures must update these values. For example, READ updates last access time, WRITE updates last modification time and change time, and LINK updates last change time.

OPEN returns a short name for a file, called a file descriptor (*fd*), which READ, WRITE, and CLOSE use to name the file. Each process starts with 3 open files: "standard in" (file descriptor 0), "standard out" (file descriptor 1), and "standard error" (file descriptor 2). A file descriptor may name a keyboard device, a display device, or a file on disk; a program doesn't need to know. This setup allows a designer to develop a program without having to worry about where the program's input is coming from and where the program's output is going to; the program just reads from file descriptor 0 and writes to file descriptor 1.

Several processes can use a file concurrently (e.g., several processes might write to the display device). If several processes open the same file, their READ and WRITE operations have each their own file cursor for that file. If one process opened a file, and then passes the file descriptor for that file to another process, then the two processes share the cursor of the file. This later case is common, because in Unix when one process (the *parent*) starts another process (the *child*), the child inherits all open file descriptors from the parent. This design allows the parent and child, for instance, to share a common output file correctly; if the child writes to the output file, for example, after the parent has written to it, the output of the child appears after the output of the parent, because they share the cursor.

If one process has a file open, and another process removes the last name pointing to that file, the inode isn't freed until the first process calls CLOSE.

To support these features, Unix extends the inode as follows:

```
structure inode
    integer block_numbers[N]   // the number of blocks that constitute the file
    integer size               // the size of the file in bytes
    integer type               // type of file: regular file, directory, symbolic link
    integer refcnt             // count of the number of names for this inode
    integer userid             // the user ID that owns this inode
    integer groupid            // the group ID that owns this inode
    integer mode               // inode's permissions
    integer atime              // time of last access (READ, WRITE,…)
    integer mtime              // time of last modification
    integer ctime              // time of last change of inode
```

To implement OPEN, READ, WRITE, and CLOSE, the file system keeps in memory several tables: one file table (*file_table*) and for each process a file descriptor table (*fd_table*). The file table records information for the files that processes have open (i.e., files for which OPEN was successful, but for which CLOSE hasn't been called yet). For each open file, this information includes the inode number of the file, its file cursor, and a reference count recording how many processes have the file open. The file descriptor table records for each file descriptor the index into the file table. Because a file's cursor is stored in the *file_table* instead of the *fd_table*, children can share the cursor for an inherited file with their parent.

With this information, OPEN is implemented as follows:

```
1    procedure OPEN (character string filename, flags, mode)
2        inode_number ← PATH_TO_INODE_NUMBER (filename, wd)
3        if inode_number = FAILURE and flags = O_CREATE then   // Create the file?
4            inode_number ← CREATE (filename, mode)            // Yes, create it.
5        else return FAILURE
6        inode ← INODE_NUMBER_TO_INODE (inode_number)
7        if PERMITTED (inode, flags) then     // Does this user have the required permissions?
8            file_index ← INSERT (file_table, inode_number)
9            fd ← FIND_UNUSED_ENTRY (fd_table)// Yes, find entry in file descriptor table
10           fd_table[fd] ← file_index                // Record file index for the file descriptor
11           return fd                        // Return fd
12       else return FAILURE                  // No, return a failure
```

Line *2* finds the inode number for the file *filename*. If the file doesn't exist, but the caller wants to create the file as indicated by the flag O_CREATE (line *3*), OPEN calls CREATE, which allocates an inode, initializes it, and returns its inode number (line *4*). If the file doesn't exist and the caller doesn't want to create it, OPEN returns a value indicating a failure (line *5*). Line *6* locates the inode. Line *7* uses the information in the inode to check if the caller has permission to open the file; the check is described in detail in section 11.6.3.4. If so, line *8* creates a new entry for the inode number in the file table, and sets the entry's file cursor to zero and reference count to 1. Line *9* finds the first unused file descriptor, records its file index, and returns it the file descriptor to the caller (lines *9* through *11*). Otherwise, it returns a value indicating a failure (line *12*).

If a process starts another process, the child process inherits the open file descriptors of the parent. That is, the information in every used entry in the parent's *fd_table* is copied to the same numbered entry in the child's *fd_table*. As a result, the parent and child entries in the *fd_table* will point to the same entry in the *file_table*, resulting in the cursor being shared between parent and child.

READ is implemented as follows:

```
1  procedure READ (fd, character array reference buf, n)
2      file_index ← fd_table[fd]
3      cursor ← file_table[file_index].cursor
4      inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number)
5      m = MINIMUM (inode.size – cursor, n)
6      atime of inode ← NOW ()
7      if m = 0 then return END_OF_FILE
8      for i from 0 to m – 1 do {
9          b ← INODE_NUMBER_TO_BLOCK (i, inode_number)
10         COPY (b, buf, MINIMUM (m – i, BLOCKSIZE))
11         i ← i + MINIMUM (m – i, BLOCKSIZE)
12     file_table[file_index].cursor ← cursor + m
13     return m
```

Lines *2* and *3* use the file index to find the cursor for the file. Line *4* locates the inode. Line *5* and *6* compute how many bytes READ can read and updates the last access time. If there are no bytes left in the file, READ returns a value indicating end of file. Lines *8* through *12* copy the bytes from the file's blocks into the caller's *buf*. Line *13* updates the cursor.

One could design a more sophisticated naming scheme for READ that, for example, allowed naming by keywords rather than by offsets; database systems typically implement such naming schemes, by representing the data as structured records that are indexed by keywords. But in order to keep its design simple, Unix restricts its representation of a file to a linear array of bytes.

The implementation of WRITE is similar to READ. The major differences are that it copies *buf* into the blocks of the inode, allocating new blocks as necessary, and that it updates the inode's *size* and *mtime*.

CLOSE frees the entry in the file descriptor table and decreases the reference count in entry in the file table. If no other processes are sharing this entry (i.e., the reference count has reached zero), it also frees the entry in the file table. If there are no other entries in the file table using this file and the reference count in the file's inode has reached zero (because another process unlinked it), then CLOSE frees the inode.

Like RENAME, some of these operations require several disk writes to complete. If the file system fails (e.g., because the power goes off) in the middle of one of the operations, then some of the disk writes may have completed and some may not. Such a failure can cause inconsistencies among the on-disk data structures. For example, the on-disk free list may show that a block is allocated but no on-disk i-node records that block in its index. If nothing is done about this inconsistency, then that block is effectively lost. Problem set *8* explores this problem and a simple, special-case solution. Chapter 9 explores systematic solutions.

Version 6 Unix (and all modern Unix implementations) maintain an in-memory cache of recently-used disk blocks. When the file system needs a block, it first checks the cache for the block. If the block is present, it uses the block from the cache; otherwise, it reads it from the storage device. With the cache, even if the file system needs to read a particular block several times, it reads that block from the storage device only once. Since reading from a disk device is often an expensive operation, the cache can improve the performance of the file system substantially. Chapter 6 discusses the implementation of caches in detail and how they can be used to improve the performance of a file system.

Similarly, to achieve high performance on operations that modify a file (e.g., WRITE), the file system will update the file's blocks in the cache, but will not force the file's modified inode and blocks to the storage device immediately. The file system delays the writes until later so that if a block is updated several times it will write the block only once, thus it can coalesce many updates in one write (see section 6.1.8).

If a process wants to ensure that the results of a write and inode changes are propagated to the device that stores the file system, it must call FSYNC; the Unix specification requires that if an invocation of FSYNC for a file returns, all changes to the file must have been written to the storage device.

2.5.12. The shell and implied contexts, search paths, and name discovery

Using the file system API, Unix implements programs for users to manipulate files and name spaces. These programs include text editors (such as *ed*, *vi* and *emacs)*, *rm* (to remove a file), *ls* (to list a directory's content), *mkdir* (to make a new directory), *rmdir* (to remove a directory), *ln* (to make link names), *cd* (to change the working directory), and *find* (to search for a file in a directory tree).

One of the more interesting Unix programs is the shell, because it illustrates a number of other Unix naming schemes. Say a user wants to compile the C source file named "x.c". The convention in Unix is to *overload* a file name by appending a suffix indicating the type of the file, such as ".c" for C source files. (A full discussion of overloading can be found in section 3.1.2.) The user types this command to the shell:

```
cc x.c
```

This command consists of two names, the name of a program (the compiler "cc") and the name of a file containing source code ("x.c") for the compiler to compile. The first thing the shell must do is find the program we want to run, "cc". To do that the Unix command interpreter uses a default context reference contained in an environment variable named PATH; that environment variable contains a list of contexts (in this case directories) in which to perform a multiple lookup for the thing named "cc". Assuming the lookup is successful, the shell launches the program, calling it with the argument "x.c".

The first thing the compiler does is try to resolve the name "x.c". This time it uses a different default context reference: the working directory. Once the compilation is underway, the file "x.c" may contain references to other named files, for example statements such as

```
#include <stdio.h>
```

This statement tells the compiler to include all definitions in the file "stdio.h" in the file "x.c". To resolve "stdio.h" the compiler needs a context in which to resolve it. For this purpose, the compiler consults another variable (typically passed as an argument when invoking the compiler), which contains a default context to be used as a search path where include files may be found. The variables used by the shell and by the compiler each consist of a series of path names to be used as the basis for an ordered multiple lookup just as was described in section 2.2.4 of this chapter.

Many other Unix programs, such as the documentation package, *man*, also do multiple lookups for files using search paths found in environment variables.

The shell resolves names for commands using the PATH variable, but sometimes it is convenient to be able to say "I want to run the program located in the current working directory". For example, a user may be developing a new version of the C compiler, which is also called "cc". If the user types "cc", the shell will look up the C compiler using the PATH variable and find the standard one instead of the new one in the current working directory.

For these cases, users can type the following command:

```
./cc x.c
```

which bypasses the PATH variable and invokes the program named "cc" in the current working directory (".").

Of course, the user could insert "." at the beginning of the PATH variable, so that all programs in the user's working directory will take precedence over the corresponding standard program. That practice, however, may create some surprises. Suppose "." is first entry in the PATH variable, and a user issues the following command sequence to the shell:

```
cd /usr/potluck
ls
```

intending to list the contents of the directory named `potluck`. If that directory contained a program named *ls* that did something different from the standard *ls* command, something surprising might happen (e.g., the program named *ls* could remove all private files)! For this reason, it is not a good idea to include names that are context-dependent, such as "." or ".." in a search path. It is better to include the absolute path name of the desired directory to the front of PATH.

Another command interpreter extension is that names can be descriptive, rather than simple names. For example, the descriptive name "*.c", matches all file names that end with ".c". To provide this extension, the command interpreter transforms the single argument into a list of arguments (with the help of a more complicated lookup operation on the entries in the context) before it calls the specified command program. In the Unix shell, users can use full-blown regular expressions in descriptive names.

As a final note, in practice, the object naming space of Unix has quite a bit of conventional structure. In particular, there are several directories with well-known names. For example, "/bin" names programs, "/etc" names configuration files, "/dev" names input/output devices, and "/usr" (rather than the root itself) names user directories. These conventions have become over time so ingrained both in programmers' minds and in programs that much Unix software will not install correctly, and a Unix wizard will become badly confused, when confronted with a system that does not follow these conventions.

2.5.13. Suggestions for further reading

For a detailed description of a more modern Unix operating system see the book describing BSD Unix [Suggestions for Further Reading 1.3.4]. A descendant of the original Unix system is Plan 9 [Suggestions for Further Reading 3.2.2], which contains a number of novel naming abstractions, some of which are finding their way back into Unix. A rich literature exists describing file system implementations and their trade-offs. A good starting point are the papers on FFS [Suggestions for Further Reading 6.3.2], LFS [Suggestions for Further Reading 9.3.1], and soft updates [Suggestions for Further Reading 6.3.3].

Exercises

Ex. 2.1. Ben Bitdiddle has accepted a job with the telephone company and has been asked to implement call forwarding. He has been pondering what to do if someone forwards calls to some number, and then the owner of that number forwards calls to a third number. So far, Ben has thought of two possibilities for his implementation:

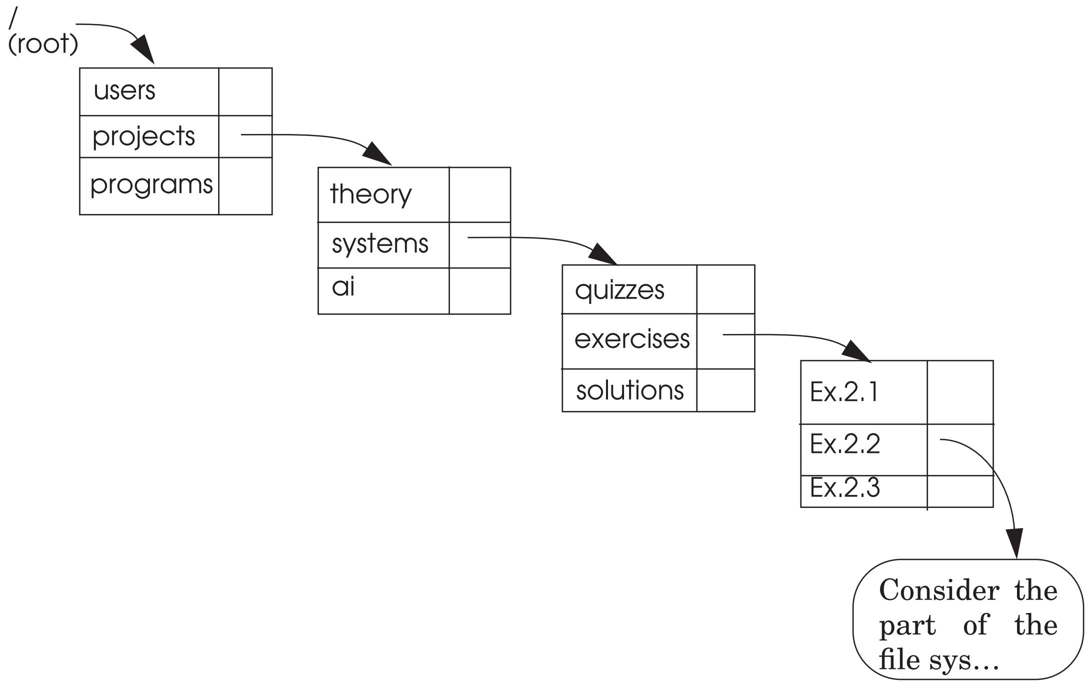
> *A.* *Follow me*. Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is baby-sitting for Bob, so she forwards her telephone to Bob. Jim calls Ann's number, Bob's telephone rings, and Ann answers it.
>
> *B.* *Delegation*. Bob is going to a party at Mary's home for the evening, so he forwards his telephone to Mary. Ann is gone for the week and has forwarded her telephone to Bob so that he can take her calls. Jim calls Ann's number, Mary's telephone rings, and Mary hands the phone to Bob to take the call.

a. Using the terminology of the naming chapter, explain these two possibilities.

b. What might go wrong if Bob has already forwarded his telephone to Mary before Ann forwards her telephone to him?

c. The telephone company usually provides *Delegation* rather than *Follow me*. Why?

Ex. 2.2. Consider the part of the file system naming hierarchy illustrated below:



You have been handed the following path name:

```
    /projects/systems/exercises/Ex.2.2
```

and you are about to resolve the third component of that path name, the name `exercises`.

a. In the path name and in the figure, identify the context that you should use for that resolution and the context reference that allows locating that context.

b. Which of the words *default*, *explicit*, *built-in*, *per-object*, and *per-name* apply to this context reference?

1995–2–1a

Ex. 2.3. One way of speeding up resolving of names is to implement a cache that remembers recently looked-up {name, object} pairs.

a. What problems do synonyms pose for cache designers, as compared with caches that don't support synonyms?

1994–2–3

b. Propose a way of solving the problems if every object has a unique ID.

1994–2–3a

Ex. 2.4. Louis Reasoner has become concerned about the efficiency of the search rule implementation in the Eunuchs system (an emasculated version of Unix). He proposes to add a *referenced object table* (ROT) which the system will maintain for each session of each user, set to be empty when the user logs in. Whenever the system resolves a name through the use a search path, it makes an entry in the ROT consisting of the name and the path name of that object. The "already referenced" search rule simply searches the ROT to determine if the name in question appears there. If it finds a match, then the resolver will use the associated path name from the ROT. Louis proposes to always use the "already referenced" rule first, followed by the traditional search path mechanism. He claims that the user will detect no difference, except for faster name resolution. Is Louis right?

1985–2–2

Additional exercises relating to chapter 2 can be found in the problem sets beginning on page PS–987.