7
FileSys: *The File System*

7.1 Chapter Objective

The objective of the FileSys project is to teach students about file-system design and organization and about the management of logical, file-based I/O in a modern operating system. To this end, students will be asked to implement the five public classes of the FileSys package: `MountTable`, which maps files to physical devices; `INode`, which keeps track of space allocation to files; `DirectoryEntry`, which defines the directory structures; `OpenFile`, which provides methods for manipulating open files via open-file handles (including the `read()` and `write()` operations); and `FileSys`, which provides a set of operations, such as `create()` and `delete()`, on non-open files.

7.2 File System Design Objectives

We briefly consider some of the main design issues in modern file systems, particularly those pertinent to the FileSys project, and then discuss how these are addressed in *OSP 2*.

Naming. In a modern file system, users are able to refer to a file by a symbolic file name. Typically such a name is in the form of a **pathname**, a sequence of directory names ending in a target file, which can also be a directory. For

example, consider the pathname `/home/fac/sas/osp/filesystem.tex`. This pathname starts at the root of the hierarchical file system directory structure (see discussion of directories below) indicated by the `/` character. The **directory separator character** `/` is also used to delimit names in the sequence. The target file in this case is `filesystem.tex`. *OSP 2* supports a hierarchical directory structure and pathnames for symbolic file naming. Pathnames can also commence from the `working directory` such as in `osp/filesystem.tex`, assuming the working directory is `/home/fac/sas`. The working directory can be manipulated interactively within the command shell.

The process of following the sequence of directory entries along a pathname to reach the target file is known as **pathname dereferencing**. Pathname dereferencing becomes more complicated by the presence of mountable file systems, discussed below.

Directory structures. Early MS-DOS file systems supported *flat* file directory structures where all files resided at the same, single level. Today's directory structures are multi-level and hierarchical where directories may contain subdirectories and so on. Such hierarchies start at the *root* directory `/`. This does not necessarily impose a tree structure on directories as files can be *linked* to from any directory, as discussed below.

Links. Modern file systems, Unix-style ones in particular, provide a `link()` system call that allows one to create a new link (directory entry) for an existing file and increment its **link count** by one. The pathname of the existing file is given as the argument to `link()`. If successful, `link()` returns the pathname of the new directory entry.

Such a directory entry is a **hard link** to the existing file, and requires that both files reside on the same file system (see discussion of mountable file systems below). Both the old and the new link share equal access and rights to the underlying object. A hard link can thus be viewed as a pointer to a file and is indistinguishable from the original directory entry. Any changes to a file are effective independent of the name used to reference the file. A hard link may not refer to directories.

A **symbolic link** is an indirect pointer to a file; its directory entry contains the name of the file to which it is linked. Symbolic links may span file systems and may refer to directories.

Mountable file systems. Another feature of modern file systems is the `mount()` system call, which requests that a removable file system be **mounted** on a specified directory. Subsequent references to this directory

will access the root directory (by default) of the mounted file system. The file system keeps track of mounted file systems and the directories on which they are mounted via a **mount table**. For example, suppose the root directory of a disk volume is mounted on `/home/fac/sas`. Then the pathname `/home/fac/sas/osp/filesystem.tex` ultimately references the target file named `filesystem.tex` on that mounted volume. Pathname dereferencing in the presence of mount tables is discussed more extensively in Section 7.4.

File storage allocation methods. How does the file system keep track of the disk blocks allocated to a particular file? Possibilities include *contiguous allocation*, where a single contiguous set of blocks is allocated to the file at the time of file creation; *chained allocation*, where each block allocated to the file contains a pointer to the next block in the chain; and *indexed allocation*, which associates a (multi-level) index structure with the file indicating the blocks that have been allocated to the file. Indexed allocation addresses many of the problems of contiguous and chained allocation, and is used in modern operating systems such as Unix, Windows, and *OSP 2*.

Free space management. How does the file system keep track of the free space on a disk, that is those disk blocks that can be allocated to a file whenever the need arises? Possibilities include *bit tables* which use a bit vector containing one bit for each block on the disk. An entry of 0 corresponds to a free block and an entry of 1 corresponds to a block in use. In the *chaining* method, each free portion of disk space contains a length field and a pointer to the next free portion in the chain. The *indexing* approach treats free space as a file and uses an index table as described under file allocation. The *free block list* method numbers each block sequentially and a list of all free blocks is maintained in a reserved portion of the disk.

7.3 Overview of the *OSP 2* File System

The *OSP 2* file system is a node-labeled tree, with support for hard links. The nodes of the tree represent files. The root node of the tree is labeled with the 1-character constant string, `FileSys.DirSeparator`, which can be "/" or "\". In the ensuing discussion, we shall use "/", but this should not be assumed in the student programs. The rest of the labels are strings of arbitrary characters except `FileSys.DirSeparator`. The labels are called **names** of files. A **full name** or the **pathname** of a file (or directory) associated with the current

node is obtained by concatenating all the labels on the path from the root to that node while separating the different names with `FileSys.DirSeparator`.

A file can be a **plain file** or a **directory**. A directory is a special file that contains information about other files. These other files are **members** of the directory; they correspond to the nodes that are children of the directory node in the tree. Thus, internal nodes of the file tree can only be directories. The leaves of the tree can be either plain files or directories. A directory that appears as a leaf is said to be **empty**.

Note that directory names that differ only in `DirSeparator` at the end are considered the same; i.e., if `DirSeparator` is "`/`" and `/foo` is a directory then `/foo/` is considered to be the same directory. Also, multiple occurrences of the separator character can be replaced by just one occurrence. For instance, `/foo/bar` and `///foo//bar` refer to the same file.

A file (or a directory) can be created and deleted. To work with a file, a thread must first **open** it and obtain an **open-file handle**. This handle contains run-time information about the file. The read and write operations are performed on the *open-file handle* rather than on the name of a file. When a thread is done working with a file, it can **close** the file handle and thus destroy it. An open-file handle is a locus of run-time information about the file. In a typical operating system it includes (among other things) the inode of the file, the task, and the current position in the file. *OSP 2* does not keep the current position, but it does maintain the rest of this information.

A pathname identifies a unique file, but a file can have any number of names. In fact, a file is uniquely represented by its **inode** (index node), which contains information about the blocks allocated to the file. Pathnames are associated with inodes through **directory entries**, but a file's inode itself contains no information about the names associated with the file. To associate another name with a file, a thread can create a **hard link** to the file, which creates another association between a pathname and the file's inode.

Deleting a file does not necessarily destroy the file's inode. Instead, it destroys the directory entry that associates the inode with a particular pathname that was used as a parameter to the `delete()` operation. Each inode has an associated **hard-link count**: the number of hard links to the inode, which is also the number of distinct names associated with the file. When a delete operation is executed on a pathname associated with a particular inode, the hard-link count is decremented by one. The inode is deleted only when *both* the hard-link count and the open count (described below) become zero.

A file's inode not only keeps track of the number of hard links to the file, but also of the file's **open count**, the number of times the file has been opened. The same inode can be open multiple times because the `open()` operation can be executed on different names associated with the file (and, in fact, even on

the same pathname). When this happens, a new open-file handle is allocated, and the same file can be accessed through different handles. Threads of the same task share the open-file handles, so typically they do not need to open the same file multiple times. However, different tasks might want to access the same file concurrently in which case they need separate file handles. When a file is opened through one of its pathnames, its open count is incremented by one. Closing a file (with the `close()` operation) decrements the open count by one.

We will now discuss each of the classes that belong to the package FileSys. Figures 7.1 and 7.2 place them in the larger context of the *OSP 2* system.

7.4 Class `MountTable`

Mount tables associate files with devices. For example, in Windows, a file named `C:\foo\bar` is said to be residing on device `C` and a file named `D:\abc\cde` is on device `D`. A mount table will then associate the letters `C` and `D` with particular physical devices.[1]

In Unix systems the association between devices and files is more flexible, but also more complex. First, Unix does not use letters to represent devices. Instead, devices are associated with directories. A mount table then is a relation that consists of a list of pairs of the form ⟨`pathname`, `deviceID`⟩. The `pathname` part of such a pair is called a **mountpoint**.

OSP 2 uses Unix-like mount tables. An example mount table is given in Figure 7.3. In that figure, we see four directories associated with four physical devices. The first question is: How does the system decide on which device any given file should reside? For example, consider the file `/foo/bar/abc/cde`. Since this file is a descendant of the root directory, `/`, and this directory is a mountpoint residing on device 0, one might think that this is where the file should live. However, this file is also in a subdirectory of mountpoint `/foo`, which lives on device 0. Looking more closely, we see that our file is also a descendant of mountpoint `/foo/bar`, which is on device 3. Which device is the correct one?

The actual mapping of files to devices works as follows. Given a full file name f, the system finds the longest name of a mountpoint d that matches f, where "matches" means that d is a prefix of f and f is a descendant of

[1] Typically a physical device is further subdivided into **partitions** and the drive letters (as well as directories in Unix—see below) are associated with partitions. In other words, partitions represent an intermediate layer between files and the actual devices they reside on. This intermediate layer does not exist in *OSP 2*, and we will ignore it here.

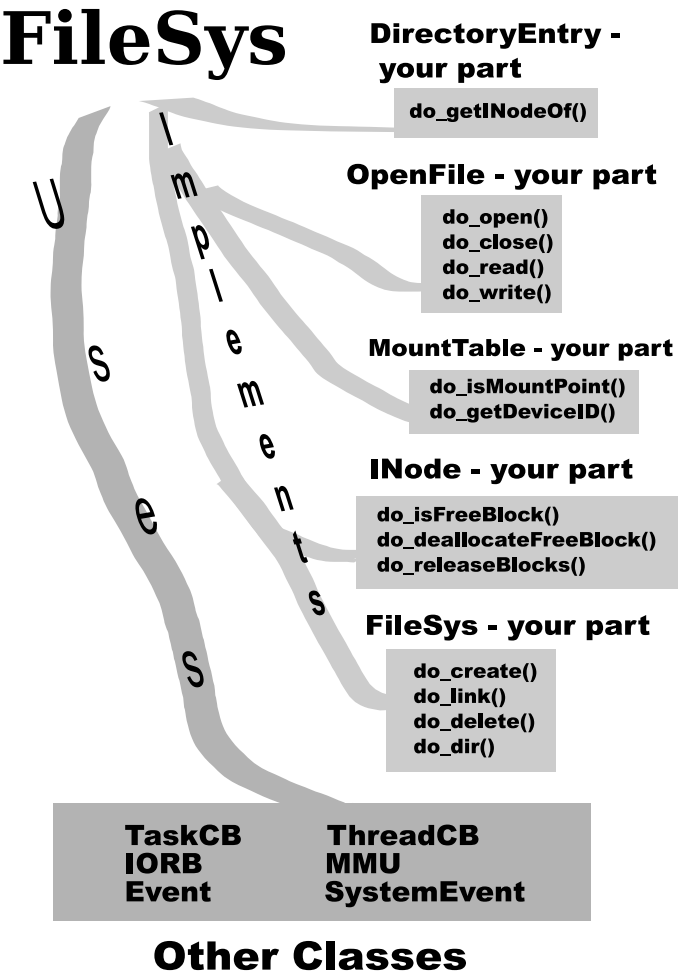

Figure 7.1 An overview of the package FileSys, I.

d in the file-tree hierarchy. For example, the longest mountpoint in the table of Figure 7.3 that matches `/foo/bar/abc/cde` is `/foo/bar` and thus the file `/foo/bar/abc/cde` resides on device 3. Note that if the mount table had a pair ⟨`/foo/bar/ab`, 4⟩ then the mountpoint `/foo/bar/ab` would *not* match `/foo/bar/abc/cde` because the latter file does *not* reside in a subdirectory of

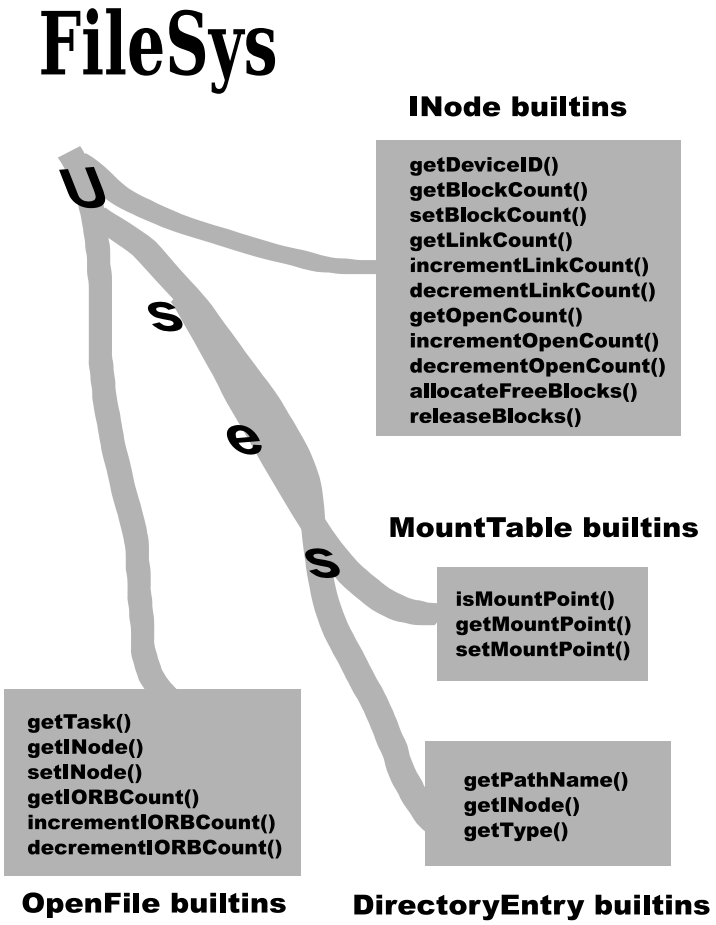

Figure 7.2 An overview of the package FileSys, II.

`/foo/bar/ab` (but rather in `/foo/bar/abc`).[2]

The `MountTable` class in *OSP 2* is intended to provide the correct mapping

[2] Another way to describe the matching criterion is to *standardize* all file names. A **standardized file name** is a full file name such that multiple occurrences of `DirSeparator` are replaced with one occurrence and if the file is a directory then `DirSeparator` is added at the end of the name. Given a file name f, the matching mountpoint is the one whose standardized name is the longest prefix of f.

Directory name	Device ID
/foo	0
/swap	2
/foo/bar	3
/	1

Figure 7.3 A mount table.

of files to devices. The mount table itself is encapsulated in a superclass of `MountTable`. What is visible, however, is the static method `getMountPoint()`, which takes a device number and returns the corresponding mountpoint. Another method, `getTableSize()`, tells the number of available physical devices (which can be different for different parameter files). Device numbers range from 0 to `getTableSize()-1`. Thus, together these methods make it possible to access all mountpoints. To provide the file-to-device mapping, the student needs to implement the following methods of class `MountTable`:

◇ `public static boolean do_isMountPoint(String dirname)`
This method tells if `dirname` is a mountpoint of one of the devices. It uses the method `getMountPoint()` internally.

◇ `public static int do_getDeviceID(String pathname)`
This method checks the mount table and returns the Id of the device that hosts the file with the given `pathname`. The method for determining the device was described earlier.

As you can see, there are no methods for creating or deleting mountpoints. In *OSP 2* all mountpoints are created by the system at startup and none gets destroyed during the execution of the system.

Built-ins and relevant methods from other classes. The implementation of these methods might need to use the following methods:

◇ `public static String getMountPoint(int deviceID)` `MountTable`
Returns the mountpoint associated with device `deviceID`. This method is an *OSP 2* built-in.

◇ `public static int getDeviceID(String pathname)` `MountTable`
Returns the device Id that hosts `pathname`. Note that this method eventually calls your method `do_getDeviceID()` described above. You have to use `getDeviceID()` here instead of `do_getDeviceID()` because of the convention explained in Section 1.9.2 that prohibits student modules from calling the `do_` methods.

◇ `final static public int getTableSize()` `Device`
Tells how many devices there are. The number of devices is specified in the parameter file and can vary from one simulation run to another.

◇ `final static public Device get(int deviceID)` `Device`
Returns the device object with the given Id. In conjunction with `getTableSize()`, this method can be used in a loop to examine each device in turn, as device IDs range from 0 to `getTableSize()-1`. Note that all devices are mounted by *OSP 2* at the beginning of the simulation and no devices are added or removed during a simulation run. Therefore the number of devices remains constant and the device table has no "holes".

Summary of the class `MountTable`

This class maintains the mount table data structure, which maintains the correspondence between devices and directories through which these devices are accessed by the programs. Other modules of the file system layer access the mount table mainly using the methods `getMountPoint()` and `getDeviceID()`.

7.5 Class `INode`

An *OSP 2* inode represents a concrete file. An inode records information about the device where the file lives, and it keeps track of the blocks occupied by the file, the hard link count, and the open count.

The most important information here is the set of blocks occupied by the file. The actual data structure to be used to capture this information is up to the student implementation, although the course instructor may have specific requirements for this data structure.

The following methods of class `INode` are to be implemented as part of the FileSys project:

◇ `public INode(int deviceID)`
The constructor. It should call `super(deviceID)` and then initialize the instance variables of the inode (if necessary).

◇ `public static boolean do_isFreeBlock(int block, int deviceID)`
Tells whether `block` on device with Id `deviceID` is free.[3]

[3] Note that from an object-oriented design perspective, this method better fits in class `Device`. However, space management is not a function of the basic I/O

◇ `public int do_allocateFreeBlock()`
When applied to an inode object, allocates a free block to that inode and returns the block number of that block. Marks the block as used. Make sure that the INode block count is set correctly (see the method `setBlockCount()`). Returns `NONE` if the device has no free blocks.

◇ `public void do_releaseBlocks()`
Releases all disk blocks occupied by the inode. Make sure that the INode block count is set correctly (`setBlockCount()`).

It is clear from the above that you have to keep track of the free space on the device. For some representations, such as bitmaps, it is useful to know the size of each device in blocks. The size can be obtained using the method `getNumberOfBlocks()` of the class `Device`.

Since you have to keep track of the valid inodes, you might also need to implement the **file allocation table** (or a **master file table**) thats hold these inodes.

Relevant methods defined in other classes.

◇ `final public int getNumberOfBlocks()` `Device`
Returns the total number of blocks on the device.

◇ `final static public int getTableSize()` `Device`
Returns the total number of devices in the device table (i.e., in the current simulation of the *OSP 2* system).

◇ `public final int getBlockCount()` `INode`
Returns the number of blocks allocated to this inode. This method is inherited from a superclass of `INode`.

◇ `public final void setBlockCount(int blockCount)` `INode`
Sets the number of blocks allocated to this inode. This method is inherited from a superclass.

◇ `public final int getDeviceID()` `INode`
Returns the device ID of this inode.

◇ `public static String getMountPoint(int deviceID)` `MountTable`
Returns the mountpoint of the given device.

supervisor that `Device` implements. This is an example of the tension between the layered architecture of an OS and the object-oriented design.

Summary of the class INode

The INode class has methods (implemented as built-ins) and variables which provide access to the various components of that class, as listed below:

openCount: The count of active open-file handles associated with the inode, obtained using `getOpenCount()` and changed via `incrementOpenCount()` and `decrementOpenCount()`.

hardLinkCount: The number of pathnames associated with the inode. This count is obtained via `getLinkCount()` and changed using the methods `incrementLinkCount()` and `decrementLinkCount()`.

blockCount: The number of blocks allocated to the file (the file size). This item is obtained using `getBlockCount()` and set using `setBlockCount()`.

device ID: The device Id of the inode. It can be obtained using the method `getDeviceID()`.

7.6 Class DirectoryEntry

If you were wondering how pathnames are associated with inodes, the suspense is over: this is done through directory entries defined by the class `DirectoryEntry`. A directory entry includes a pathname, an inode, and a type (`FileEntry` or `DirEntry`). The type indicates whether the particular directory entry represents a plain file or a directory.

The methods of this class to be implemented as part of the FileSys project are listed below.

◇ `public DirectoryEntry(String pathname, int type, INode inode)`
The class constructor. Calls `super()`, as usual, and initializes instance variables, if necessary.

◇ `public static INode do_getINodeOf(String pathname)`
Given a pathname, returns the corresponding inode. In order to make this possible, the class `DirectoryEntry` must maintain the collection of all directory entries.

In addition, you need to implement a number of supporting methods that other classes in your package might need to use to insert directory entries into the directories, delete the entries, etc.

Relevant builtins and methods defined in other classes. This class does not use any standard methods defined in other classes of *OSP 2*. Some standard classes provided by Java itself might be useful. For instance, `Hashtable` and the associated methods can be used to maintain the `DirectoryEntry` data structure. This would closely correspond to how directories are implemented in real operating systems.

Summary of Class `DirectoryEntry`

This class does not provide any methods, but there are several variables:

pathname: This property is accessible through the method

```
final public String getPathname()
```

This is the pathname represented by this directory entry.

INode: This property is accessible through the method

```
final public INode getINode()
```

It is the inode that this directory entry associates with the pathname of the directory entry. A related method in this class is `getINodeOf()`, which takes a pathname parameter and returns the corresponding INode:

```
final public static INode getINodeOf(String pathname)
```

Unlike `getINode()`, this method is static.

type: This property is accessible through the method

```
final public int getType()
```

It specifies the type of the directory entry, i.e., whether the entry represents a regular plain file (`FileEntry`) or a directory (`DirEntry`).

7.7 Class OpenFile

Class `OpenFile` provides methods for creating open-file handles, accessing the components of an open-file handle, and using open-file handles to perform I/O operations.

◇ `public OpenFile(INode inode, TaskCB task)`
This is a constructor for open-file handles. It must call `super()` with the same set of parameters and then, possibly, initialize the various variables that you might have added to the class.

◇ `static public OpenFile do_open(String filename, TaskCB task)`
This method create an open-file handle. It receives a file name (which must correspond to a previously created file) and a task object, creates an open-file handle for the file, and adds the handle to the task's table of open files. (Recall from Chapter 3 that the open-files table is one of the resources owned by a task.)

First, the file must already exist before it can be opened. Existence should be checked using a method that you implement in class `FileSys`. Note that this method will be unknown to the *OSP 2* IFL layer, i.e. it will not have a wrapper method in the IFL, and therefore its implementation and name are completely up to you. Second, opening a mountpoint is a violation, so you must check that the argument is not a mountpoint. (The method `isMountPoint()` of class `MountTable` can be used to check this.)

Once you pass these checks, a new open-file handle can be created. The `OpenFile()` constructor takes an inode and a task as parameters, so you must obtain the inode corresponding to `filename` (using the method `getINodeOf()` discussed earlier). After constructing the handle, you should add it to the task with the method `addFile()` of class `TaskCB`. Finally, the count of open files for the inode should be incremented (`incrementOpenCount()`) and the newly created file handle returned.

◇ `public int do_close()`
A file is closed when its open-file handle is no longer needed. However, closing a file is trickier than it might seem.

First, the file might still have outstanding (unprocessed) IORBs. As discussed in Chapter 6, such a file cannot be closed immediately. Instead, you should *mark* the file as needing to be closed later and leave it alone. Marking is performed by setting the `closePending` flag to `true`, where `closePending` is a field of the context `OpenFile` object. The disk interrupt handler will close the file (by issuing another `close` operation) after the last outstanding IORB has been processed.

If the file *cannot* be closed due to outstanding IORBs, as described above, `do_close()` should just exit and return `FAILURE`. If the file *can* be closed immediately, then you should do so, adjusting the relevant structures. One thing that needs to be done here is to decrement the open file count of the inode associated with the file handle. The inode is obtained using the `getINode()` method and the count is changed using `decrementOpenCount()` of class `INode`.

Next, you should check whether you can destroy the inode associated with the file handle and release the disk blocks owned by that inode. As discussed earlier, an inode can be deleted when both its open file count (`getOpenCount()`) and its hard-link count (`getLinkCount()`) are zero. The inode's disk blocks are released with the method `releaseBlocks()` of class `INode`. The method to remove an inode from the disk master file table should reside in class `INode` and its name (and, of course, its implementation) are left for you to decide.

Finally, the `closePending` field is reset to false, the file handle is removed from the open-files table of the task associated with that handle, and `SUCCESS` is returned.

◇ `public int do_read(int fileBlockNumber, PageTableEntry memoryPage, ThreadCB thread)`
The `do_read()` method is executed on a file-handle object. It creates a read request to the device associated with the file handle, enqueues the request to the device, and waits until the I/O is complete — I/O operations in *OSP 2* are synchronous at the thread level. That is, the thread that issues an I/O operation is eventually blocked until the operation is finished.[4]

It is recommended that you make sure that the parameters passed to `open()` are consistent. For example, the `fileBlockNumber` parameter must be within the appropriate range (non-negative and not exceed the file size). If it is not, `FAILURE` should be returned. Likewise, it is wise to check whether `memoryPage` and `thread` are not null.

In the next step, a new system event is created using the constructor `SystemEvent()` and the current thread is suspended on that event. At this point it is recommended that you refresh your memory about thread suspension and resumption by (re-)reading Section 4.3. A thread that is suspended on a system event is not really blocked, but instead can be thought of as

[4] However, I/O is asynchronous at the task level: a thread that does not wish to wait for I/O can spawn another thread that performs the I/O. Meanwhile, the first thread can go about its business while the second thread would wait. When the I/O is done, the two threads can merge.

having changed status from user thread to system thread. When the read operation is complete, the event will "happen" and the thread will be resumed. To be able to resume the thread after the I/O is complete, you should save the `SystemEvent` object in a variable.

You are now ready to construct an IORB for the request. The inode and device Id can be extracted from the open-file handle using the appropriate methods. The I/O type (one of the parameters in the IORB constructor) is, naturally, `FileRead`. The only thing that requires care is the disk block number parameter to the constructor.

Note that the `fileBlockNumber` parameter to `do_read()` is the number of the *logical* block within a file. It must be mapped to the *physical* block of the disk. Information about the disk blocks allocated to the file is stored in the inode, which is implemented in your `INode` class. It is recommended that you implement a method in `INode` that, when applied to an inode with a logical file block number as a parameter, returns the corresponding physical block.

After collecting all the needed components, you use the `IORB()` constructor to create an IORB for the read request.

Next, you must enqueue the request to the appropriate device using the method `enqueueIORB()` of class `Device`. Note that `enqueueIORB()` locks the target memory buffer page, which can cause some swapping activity, and the thread must wait until swapping is finished. As usual in *OSP 2*, a waiting thread might get killed, so it is necessary to ascertain that the thread is still alive after `enqueueIORB()` returns. If the thread was killed, `do_read()` should return `FAILURE`.

If `enqueueIORB()` finished successfully, `thread` must be suspended on `iorb`. When this I/O completes, `thread` will be notified and control will get past the `suspend()` operation. At this point, again, you must check if the thread is still alive. If it is dead, `FAILURE` is returned; if it is alive, you execute `notifyThreads()` on the previously created `SystemEvent` object and return `SUCCESS`.[5]

◇ `public int do_write(int fileBlockNumber, PageTableEntry memoryPage, ThreadCB thread)`

Writing is similar to reading in many respects. One important difference (in *OSP 2*, anyway) is that a file block is considered out of range only if it is

[5] Note that the logic of your implementation should be such that each `suspend()` is matched by a `notifyThreads()` system call.

negative. If `fileBlockNumber` is higher than the number of blocks in the file, the file is extended with the necessary number of blocks. For example, if the current size of the file is 2 blocks and `fileBlockNumber` is 5, then 4 new blocks must be allocated to the file. (Note that blocks are counted from 0, so 5 refers to the 6th block of the file.) Additional disk blocks are allocated to an inode as a result of the `allocateFreeBlock()` system call (*and not by any other means*!).

Another important difference is that the device might not have enough free space to accommodate the file expansion. In this case, `FAILURE` should be returned. Note that free disk space management is done in class `INode` and is the student's responsibility.

Relevant methods defined in other classes.

◇ `public static boolean isMountPoint(String dir)` — `MountTable`
Tells if `dir` is a mountpoint.

◇ `final public void addFile(OpenFile file)` — `TaskCB`
Adds `file` to the open-files table of the task.

◇ `final public void removeFile(OpenFile file)` — `TaskCB`
Removes the file handle from the task's open files table.

◇ `final public void suspend(Event event)` — `ThreadCB`
Suspends thread on the event.

◇ `public void notifyThreads()` — `Event`
Notifies threads that are waiting on the event.

◇ `final public int getIORBCount()` — `OpenFile`
Returns the IORB count of the open-file handle.

◇ `final public void incrementIORBCount()` — `OpenFile`
Increments the IORB count of the open-file handle by 1.

◇ `final public void decrementIORBCount()` — `OpenFile`
Decrements the IORB count of the open-file handle by 1.

◇ `final public INode getINode()` — `OpenFile`
Returns the inode of the open-file handle.

◇ `final public void setINode(INode inode)` — `OpenFile`
Sets the inode of the open file handle.

◇ `final public TaskCB getTask()` — `OpenFile`
Returns the task of the open-file handle.

◇ `public final int getOpenCount()` `INode`
Returns the open file count of inode.

◇ `public final void incrementOpenCount()` `INode`
Increments the open-file count of the inode by 1.

◇ `public final void decrementOpenCount()` `INode`
Decrements the open file count of inode by 1.

◇ `final public void releaseBlocks()` `INode`
Frees up disk blocks held by the inode.

◇ `public SystemEvent(String type)` `SystemEvent`
The constructor for system events. The `type` parameter is used to provide a tag with which the event will be displayed in the log file. This tag can be useful for debugging when you need to trace the execution of your project. When a thread is suspended on a `SystemEvent`, it can be thought of as having changed status from user thread to system thread. See Section 4.3 for more details on suspension and resumption of threads.

◇ `public IORB(ThreadCB thread, PageTableEntry page, int blockNumber, int deviceID, int ioType, OpenFile openFile)`
Creates an IORB with the given parameters.

◇ `final public int enqueueIORB(IORB iorb)` `Device`
Enqueues `iorb` to its associated device. This operation is blocking and can cause a pagefault (and the ensuing swapping) because `enqueueIORB()` needs to lock the target memory page in order to shield it from page replacement. See Chapters 5 and 6 for a more thorough explanation of page locking. This method returns `SUCCESS` if `iorb` has been successfully enqueued. A failure is returned when enqueuing fails (for example, if the original thread has died).

◇ `final public int allocateFreeBlock()` `INode`
Allocates a free block to the inode. The block becomes occupied.

Summary of the class `OpenFile`

The class `OpenFile` maintains the following important variables, which are affected using the various methods of that class.

IORB count: The number of outstanding IORBs for the handle. Obtained using `getIORBCount()` and changed using `incrementIORBCount()` and `decrementIORBCount()`.

INode: The inode of the open-file handle. Obtained using `getINode()` and set using `setINode()`.

Task: The task that owns the open-file handle. Obtained using the `getTask()` method.

closePending: This field is set to true by `do_close()` if the `OpenFile` object has outstanding IORBs and cannot be closed immediately. When the last IORB for this `OpenFile` object is processed, `do_close()` will close the file.

7.8 Class `FileSys`

You are to implement the following methods of class `FileSys` as part of this project.

◇ `public static void init()`
As usual in *OSP 2*, this method is called at the beginning of every simulation run. It can be used to initialize static variables that your implementation might use (for example, the variables used in the implementation of the mount table, in the open-files table, in the list of free blocks on the various devices, etc.).

◇ `final static public int do_create(String pathname, int size)`
This method creates a file with a given `pathname` and `size` (in bytes). In one sentence, this means making the necessary checks and then creating the corresponding inode and the directory entry that relates `pathname` with that inode. The devil is in the details, however, and this is what we will be discussing next.

First, you have to check if the file with the same name already exists. If so, `FAILURE` is returned. If a file is a mountpoint (is listed in the mount table), then it is presumed to exist right from the start and, since mountpoints cannot be created or destroyed, `FAILURE` should be returned in this case as well. If the file does not exist, check if `pathname` refers to a directory or a plain file. A pathname refers to a directory if it ends with the filename separator, `DirSeparator`, but is not a mountpoint. It refers to a plain file otherwise.

Note, however, that the convention that a directory name must end with `DirSeparator` is used in the `create()` call only (just in order to avoid introducing yet another system call). In all other contexts, pathnames such as `/foo/bar` and `/foo/bar/` refer to the same directory. Also, if a plain file by the name `/foo/bar` already exists and `do_create()` is called with

/foo/bar/ as a parameter, the call should fail and FAILURE returned, because there *cannot* be a file and a directory with the same name. Likewise, if do_create("/foo/bar/",...) was earlier called to create a directory, then a subsequent call do_create("/foo/bar",...) should fail, because otherwise we would have a file and a directory with the same name.

In view of the above, it is generally a good idea to **normalize** file names before doing any file-name comparisons. A **normalized pathname** is a full pathname such that it does not have repeated occurrences of DirSeparator (pathnames /foo///bar// and /foo/bar/ are considered the same, but only the latter is normalized). It may be convenient to also remove the trailing DirSeparator in normalized directory names (except for the root of the file system, /), but this depends on the particular algorithms that you are using.

Next, you must check if the caller intended to create a file or a directory by checking the last character of pathname. The appropriate file-type indicator (FileEntry or DirEntry) will later go into the directory entry for the file. Also, for plain files, the size parameter indicates the size of the file in bytes. However, for directories this parameter is ignored, since directories are assumed to occupy *exactly one* disk block. The correct size parameter should be used when constructing the corresponding inode.

It is common in programming to attempt to create a file in a non-existent directory with the intent that the system would create all the intermediate subdirectories automatically. For instance, suppose that the directory /foo exists, but /foo/bar does not. In *OSP 2*, the call do_create("/foo/bar/moo/abc.html",...) should then create the intermediate directories, /foo/bar and /foo/bar/moo, before creating /foo/bar/moo/abc.html. Note that this means that while creating the intermediate directories, do_create() will call create() (its *OSP 2* wrapper), which in turn will call do_create() *recursively*.

Next you should check the mount table to determine the device where the file is to be created. Recall from Section 7.4 that determining the device is the job of method getDeviceID() of class MountTable. You need to make sure that the device has enough free space. Recall that space management is the job of the INode class. You might want to implement a method in that class which returns the number of free blocks. If this number is less than the number of blocks needed to accommodate our file, FAILURE should be returned. It is therefore important to correctly calculate the number of blocks needed to accommodate a file-creation request. Recall that do_create() gets the size of the file in bytes, and this has to be converted into disk blocks. The block size equals the size of a virtual memory page, which can be obtained using the two methods provided by the class MMU: getVirtualAddressBits() and

`getPageAddressBits()`.[6]

Note, however, that *OSP 2* assumes that directories occupy exactly one block and the file-size parameter in `do_create()` should be ignored in this case.

After all these checks, nothing (but a computer crash) can stop us from creating the file. You can use the constructor for the class `INode` to create a new inode. Next, you should use methods `incrementLinkCount()` and `allocateFreeBlock()` of `INode` to update the count of hard links to the inode and to allocate the right number of disk blocks to it. The inode should also be inserted into the device's file allocation table for safekeeping.

To complete the process, you must create a directory entry for `pathname` and insert it into the appropriate directory. This is accomplished using the constructor of `DirectoryEntry` and other methods that depend on your implementation of directories.

When all is done, `SUCCESS` is returned.

◇ `final static public int do_link(String pathname, String linkname)`

This method creates a new hard link, with name `linkname`, to the inode associated with `pathname`. The process is similar to creating a file: you need to check if a directory entry for `linkname` already exists and return `FAILURE` if it does. Otherwise (if there is no file named `linkname`), you must create an appropriate directory entry. However, there also are significant differences between linking and creating files.

First, no new inode need be created. Instead, the inode associated with `pathname` is used. Therefore, no additional space need be allocated. Second, hard links to directories are not allowed (as in Unix). Third, unlike the case of file creation, no intermediate directories are created. So, if the directory `/foo` exists but `/foo/bar` does not, then creation of a hard link `/foo/bar/abc.html` to another file should fail.

Other than that, creation of a new directory entry to associate `linkname` with the inode of `pathname` proceeds as in the case of `do_create()`. In particular, do not forget to increment the hard-link count.

Note one interesting thing: after a hard link to an inode is created, `linkname` and `pathname` become virtually indistinguishable. That is, `linkname` is as much of a "file name" for the corresponding inode as `pathname` is. The inode

[6] Note that a file-creation request might specify size 0, in which case the request must succeed even if the device has no room.

itself does not contain any file-name information and all the naming takes place in directory entries.

◇ `final static public int do_delete(String pathname)`
Destroying a file is not as simple as it might seem. First, you must check if a file with the name `pathname` exists. Note that you cannot always tell from the name whether it refers to a plain file or a directory, so you must use normalized names to do the checks. Also, non-empty directories cannot be deleted and, of course, deletion of mountpoints is not allowed. In all these cases, `FAILURE` should be returned.

Once you get past these checks, you must remember that `pathname` is just one of the several possible hard links to the inode associated with a file. If after deleting the directory entry for `pathname` and decrementing the hard-link count the number of hard links for the inode (obtained via `getLinkCount()`) is non-zero, do not delete the inode. Recall that inodes also have an open count, in addition to a hard-link count, which counts the number of open-file handles for the inode. If this count is positive, the inode must *not* be deleted. In both cases, however, the directory entry for `pathname` must still be deleted. If the hard-link count as well as the open count are zero, both the inode and the directory entry must be deleted. In case the inode is deleted, all its blocks must be freed up (using `releaseBlocks()`). Finally, `SUCCESS` should be returned.

◇ `final static public Vector do_dir(String dirname)`
This method returns a vector of *normalized* file names that reside in directory `dirname`. If `dirname` does not exist or is not a directory, `null` is returned.

Relevant methods from other classes. The following methods might be required to implement class `FileSys`.

◇ `public static boolean isMountPoint(String dir)` `MountTable`
Tells if a given pathname is a mountpoint.

◇ `static final public int getVirtualAddressBits()` `MMU`
Tells how many bits are used to represent a virtual address. This method and the next method can be used to determine how many bits are needed to represent an address within a page, from which the page/block size can be computed.

◇ `static final public int getPageAddressBits()` `MMU`
Tells the number of bits used to represent a page address.

◇ `public final int getLinkCount()` `INode`
Returns the number of hard links to the inode.

◇ `public final void decrementLinkCount()` — `INode`
Decrements the hard-link count for inode.

◇ `public final void incrementLinkCount()` — `INode`
Increments the hard-link count for the inode.

◇ `public final int getOpenCount()` — `INode`
Returns the count of open-file handles for the inode.

◇ `final public int allocateFreeBlock()` — `INode`
Allocates a free block to the inode. The block becomes occupied.

◇ `final public void releaseBlocks()` — `INode`
Releases all the blocks held by the inode.

◇ `public final int getDeviceID()` — `INode`
Tells the device Id of the inode.

◇ `final public static int create(String name, int size)` — `FileSys`
The *OSP 2* wrapper for `do_create()`

◇ `final public static INode getINodeOf(String pathname)` — `DirectoryEntry`
Returns the inode associated with `pathname`. If no directory entry for `pathname` exists, returns `null`.

◇ `final public static void showDirectory(String dirname)` — `DirectoryEntry`
Prints the directory listing for `dirname` to the log file. This method can be useful for debugging, since it shows what *OSP 2* believes the correct listing is supposed to be.

Summary of Class `FileSys`

In *OSP 2*, the class `FileSys` does not typically maintain important data structures of its own. Instead, it serves as a container for methods that do not logically belong to any other class in the package. For instance, the method `do_delete()` for deleting files based on a string that represents the file name cannot be naturally attached to any other *OSP 2* class. Such methods do not normally maintain complex data of its own. Instead, they operate on the data structures defined in other classes, such as `DirectoryEntry` or `MountTable`, using the methods provided in those classes.

7.9 Methods Exported by the FileSys Package

The following is a summary of the public methods defined in the classes of the FileSys package or in the corresponding superclasses, which can be used to implement this and other student packages. To the right of each method we list the class of objects to which the method applies.

◇ `final public static int create(String name, int size)` FileSys
Creates a file with the specified name and size.

◇ `final public static void delete(String name)` FileSys
Deletes the directory entry for the specified file.

◇ `final public static OpenFile open(String filename, TaskCB task)` OpenFile
Opens the specified file `filename` by `task` and returns the newly created open-file handle (or `null`, if the operation fails).

◇ `final public int close()` OpenFile
Closes the file handle on which this operation is invoked.

◇ `final public void read(int fileBlockNumber, PageTableEntry memoryPage, ThreadCB thread)` OpenFile
Performs the read I/O operation using the given open-file handle. Reads data from logical file block `fileBlockNumber` into `memoryPage` on behalf of `thread`.

◇ `final public void write(int fileBlockNumber, PageTableEntry memoryPage, ThreadCB thread)` OpenFile
Performs the write I/O operation using the given open-file handle. Writes data to logical file block `fileBlockNumber` from `memoryPage` on behalf of `thread`.