MOSS File System Simulator
Installation on Unix/Linux/Solaris/HP-UX Systems

Purpose

This document provides instructions for the installation of the MOSS File System Simulator on Unix operating systems. This procedure should be the same or similar on Unix, Linux, Solaris, HP-UX and other Unix-compatible systems.

The MOSS software is designed for use with Andrew S. Tanenbaum, Modern Operating Systems, 2nd Edition (Prentice Hall, 2001). The File System Simulator and documentation were written by Ray Ontko (*rayo@ontko.com*).

Requirements

The following software components are required to install and use the MOSS File System Simulator.

- Java Development Kit (JDK) 1.0 or greater
- Text program editor (e.g., notepad)

Installation

Installation of the software can be accomplished with these simple steps:

1. Create a directory in which you wish to install the simulator (e.g., "moss/filesys").

```
cd
mkdir moss
cd moss
mkdir filesys
cd filesys
```

2. Download the compressed tar archive (filesys.tgz) into the directory.

3. Expand the compressed tar archive.

```
$ tar -xzf filesys.tgz
```

or

```
$ gunzip filesys.tgz
$ tar xf filesys.tar
```

Files

The directory should now contain the following files:

Files	*Description*
filesys.tgz	Compressed tar archive which contains all the other files.
BitBlock.java Block.java DirectoryEntry.java FileDescriptor.java FileSystem.java IndexNode.java Kernel.java ProcessContext.java Stat.java SuperBlock.java cat.java cp.java dump.java ls.java mkdir.java mkfs.java tee.java	Java source files (*.java)
BitBlock.class Block.class DirectoryEntry.class FileDescriptor.class FileSystem.class IndexNode.class Kernel.class ProcessContext.class Stat.class SuperBlock.class cat.class cp.class dump.class ls.class mkdir.class mkfs.class tee.class	Compiled Java class files (*.class)
install_unix.html install_windows.html user_guide.html	Documentation
javadoc	Directory containing documentation on java classes (generated by the javadoc utility).
copying.txt	Gnu General Public License: Terms and Conditions for Copying, Distribution, and Modification

Compilation

The distribution includes compiled class files as well as the source java files. You should not need to recompile unless you decide to change the code. If you wish to compile the code, the following commands should work if you're using a Java compiler that accepts the normal "javac" command line.

To determine which shell you're using:

```
$ echo $SHELL
```

If you're using sh, ksh, bash:

```
$ CLASSPATH=.
$ export CLASSPATH
$ javac *.java
```

If you're using csh, tcsh:

```
% set CLASSPATH=.
```

% javac *.java

Test

To test the program, enter the following sequence of commands.

```
$ java mkfs filesys.dat 256 16
$ java mkdir /root
$ java ls /
$ ls *.java | java tee /root/t.lis
$ java ls /root
$ java cat /root/t.lis
```

If things are working correctly, the program will produce output that looks something like this:

```
$ java mkfs filesys.dat 256 16
block_size: 256
blocks: 16
super_blocks: 1
free_list_blocks: 1
inode_blocks: 3
data_blocks: 11
block_total: 16

$ java mkdir /root

$ java ls /

/:
   0     48 .
   0     48 ..
   1     32 root
total files: 3

$ ls *.java | java tee /root/t.lis
BitBlock.java
Block.java
DirectoryEntry.java
FileDescriptor.java
FileSystem.java
IndexNode.java
Kernel.java
ProcessContext.java
Stat.java
SuperBlock.java
cat.java
cp.java
dump.java
ls.java
mkdir.java
mkfs.java
tee.java

$ java ls /root

/root:
   1     48 .
   0     48 ..
   2    219 t.lis
total files: 3

$ java cat /root/t.lis
BitBlock.java
Block.java
DirectoryEntry.java
FileDescriptor.java
FileSystem.java
IndexNode.java
Kernel.java
ProcessContext.java
Stat.java
SuperBlock.java
cat.java
cp.java
dump.java
ls.java
mkdir.java
mkfs.java
tee.java
```

MOSS File System Simulator
User Guide

Purpose

This document is a user guide for the MOSS File System Simulator. It explains how to use the simulator and describes the programs and the various input files used by and output files produced by the simulator. The MOSS software is designed for use with Andrew S. Tanenbaum, Modern Operating Systems, 2nd Edition (Prentice Hall, 2001). The File System Simulator and documentation were written by Ray Ontko (*rayo@ontko.com*).

Introduction

The file system simulator shows the inner workings of a UNIX V7 file system. The simulator reads or creates a file which represents the disk image, and keeps track of allocated and free blocks using a bit map. A typical exercise might be for students to write a program (in Java) which invokes various simulated operating system calls against a well-known disk image provided by the instructor. Students may also be asked to implement indirect blocks, list-based free block managment, or write a utility (like fsck) to check and repair the file system.

Overview

The MOSS File System Simulator is a collection of Java classes which simulate the file system calls available in a typical Unix-like operating system. The "Kernel" class contains methods (functions) like "creat()", "open()", "read()", "write()", "close()", etc., which read and write blocks in an underlying file in much the same way that a real file system would read and write blocks on an underlying disk device.

In addition to the "Kernel" class, there are a number of underlying classes to support the implementation of the kernel. The classes FileSystem, IndexNode, DirectoryEntry, SuperBlock, Block, BitBlock, FileDescriptor, and Stat contain all data structures and algorithms which implement the simulated file system.

Also included are a number of sample programs which can be used to operate on a simulated file system. The Java programs "ls", "cat", "mkdir", "mkfs", etc., perform file system operations to list directories, display files, create directories, and create (initialize) file systems. These programs illustrate the various file system calls and allow the user to carry out various read and write operations on the simulated file system.

As mentioned above, there is a backing file for our simulated file system. A "dump" program is included with the distribution so that you can examine this file, byte-by-byte. Any dump program may be used (e.g., the "od" program in Unix); we include this one which is simple to use and understand, and can be used with any operating system.

There are a number of ways you can use the simulator to get a better understanding of file systems. You can

- use the provided utility programs (mkfs, mkdir, ls, cat, etc.) to perform operations on the simulated file system and use the dump program to examine the underlying file and observe any changes,
- examine the sample utility programs to see how they use the system call interface to perform file operations,
- enhance the sample utility programs to provide additional functionality,
- write your own utility programs to extend the functionality of the simulated file system, and
- modify the underlying Kernel and other implementation classes to extend the functionality of the

In the sections which follow, you will learn what you need to know to perform each of these activities.

Using File System Simulator Programs

Using mkfs

The mkfs program creates a file system backing file. It does this by creating a file whose size is specified by the block size and number of blocks given. It writes the superblock, the free list blocks, the inode blocks, and the data blocks for a new file system. Note that it will overwrite any existing file of the name specified, so be careful when you use this program.

This program is similar to the "mkfs" program found in Unix-like operating systems.

The general format for the mkfs command is

> java mkfs *file-name block-size blocks*

where

file-name
: is the name of the backing file to create (e.g., filesys.dat). Note that this is the name of a real file, not a file in simulator. This is the file that the simulator uses to simulate the disk device for the simulated file system. This may be any valid file name in your operating system environment.

block-size
: is the block size to be used for the file system (e.g., 256). This should be a multiple of the index node (i-node) size (usually 64) and the directory entry size (usually 16). Modern operating systems usually use a size of 1024, or 512 bytes. We use 128 or 256 byte block sizes in many of our examples so that you can quickly see what happens when directories grow beyond one block. This should be a decimal number not less than 64, but less than 32768.

blocks
: is the number of blocks to create in the file system(e.g., 40). This number includes any blocks that may be used for the superblock, free list management, inodes, and data blocks. We use a relatively small number here so that you can quickly see what happens if you run out of disk space. This can be any decimal number greater than 3, but not greater than 2^{24} - 1 (the maximum number of blocks), although you may not have sufficient space to create a very large file.

For example, the command

```
java mkfs filesys.dat 256 40
```

will create (or overwrite) a file "filesys.dat" so that it contains 40 256-byte blocks for a total of 10240 bytes.

The output from the command should look something like this:

```
block_size: 256
blocks: 40
super_blocks: 1
free_list_blocks: 1
inode_blocks: 8
data_blocks: 30
block_total: 40
```

From the output you can see that one block is needed for the superblock, one for free list management, eight for index nodes, and the remaining 30 are available for data blocks.

Why is there 1 block for free list management? Note that 30 blocks require 30 bits in the free list bitmap. Since 256 bytes/block * 8 bits/byte = 2048 bits/block, clearly one bitmap block is sufficient to track block allocation for this file system.

Why are there 8 blocks for index nodes? Note that 30 blocks could result in 30 inodes if many one-block files or directories are created. Since each inode requires 64 bytes, only 4 will fit in a block. Therefore, 8 blocks are set aside for up to 32 inodes.

Using mkdir

The mkdir program can be used to create new directories in our simulated file system. It does this by creating the file specified as a directory file, and then writing the directory entries for "." and ".." to the newly created file. Note that all directories leading to the new directory must already exist.

This program is similar to the "mkdir" command in Unix-like and MS-DOS-related operating systems.

The general format for the mkdir command is

> java mkdir *directory-path*

where

directory-path
: is the path of the directory to be created (e.g., "/root", or "temp", or "../home/rayo/moss/filesys"). If *directory-path* does not begin with a "/", then it is appended to the path name for working directory for the default process.

For example, the command

```
java mkdir /home
```

creates a directory called "home" as a subdirectory of the root directory of the file system.

Similarly, the command

```
java mkdir /home/rayo
```

creates a directory called "rayo" as a subdirectory of the "home" directory, which is presumed to already exist as a subdirectory of the root directory of the file system.

Using ls

The ls program is used to list information about files and directories in our simulated file system. For each file or directory name given it displays information about the files named, or in the case of directories, for each file in the directories named.

This program is similar to the "ls" command in Unix-like operating systems, or the "dir" command in DOS-related operating systems.

The general format for the ls command is

> java ls *path-name ...*

where

path-name ...
: is a space-separated list of one or more file or directory path names.

For example, the command

```
java ls /home
```

lists the contents of the "/home" directory. For each file in the directory, a line is printed showing the name of the file or subdirectory, and other pertinent information such as size.

The output from the command should look something like this:

```
/home:
   1     48 .
   0     48 ..
   2     32 rayo
total files: 3
```

In this case we see that the "/home" directory contains entries for ".", "..", and "rayo".

Using tee

The tee program reads from standard input and writes whatever is read to both standard output and the named file. You can use this program to create files in our simulated file system with content created in the operating system environment.

This program is similar to the "tee" command found in many Unix-like operating systems.

The general format for the tee command is

> java tee *file-path*

where

file-path
: is the name of a file to be created in the simulated file system. If the named file already exists, it will be overwritten.

For example,

```
echo "howdy, podner" | java tee /home/rayo/hello.txt
```

causes the single line "howdy, podner" to be written to the file "/home/rayo/hello.txt".

The output from the command is

```
howdy, podner
```

which you should note was the same as the input sent to the tee program by the "echo" command.

Note that the "|" (pipe) is almost always used with the tee program. Users of Unix-like operating systems will find the "echo", and "cat" commands useful to produce input for the pipe to tee. Users of MS-DOS-related operating systems will find the "echo" and "type" commands to be useful in this regard.

If you wish to simply enter text directly to a file, then you may use tee directly (i.e., without the pipe). Users of Unix-like operating systems will need to use CTRL-D to signal the end of input. Users of MS-DOS-related operating systems will need to use CTRL-Z to signal the end of input.

Using cp

The cp program allows you to copy the contents from one file to another in our simulated file system. If the destination file already exists, it will be overwritten.

This program is similar to the "cp" command in Unix-like operating systems, and the "copy" command in MS-DOS-related operating systems.

The general format of the "cp" command is

> java cp *input-file-name output-file-name*

where

input-file-name
: is the path-name for the file to be copied (i.e., the *source* file, and

output-file-name
: is the path-name for the file to be created (i.e., the *target* file.

For example,

```
java cp /home/rayo/hello.txt /home/rayo/greeting.txt
```

creates a new file "/home/rayo/greeting.txt" by copying to it the contents of file "/home/rayo/hello.txt".

Using cat

The cat program reads the contents of a named file and writes it to standard output. The cat program is generally used to display the contents of a file.

This program is similar to the "cat" command in Unix-like operating systems, or the "type" command in MS-DOS-related operating systems.

The general format of the cat command line is

> java cat *file-name*

where

file-name
: is the name of the file from which data are to be read for writing to standard output.

For example,

```
java cat /home/rayo/greeting.txt
```

causes the file "/home/rayo/greeting.txt" to be read, the contents of which are written to standard output.

In this case, the output from the program might look something like this

```
howdy, podner
```

Dumping the File System

While you are working with the file system simulator, you may wish to dump the contents of the backing file to *see* if it contains what you *think* it contains. The dump program shows the contents of a file in the operating environment, one byte at a time, in various formats (hexadecimal, decimal, ASCII).

Note that dump dumps the contents of a real file, not a file in our simulated file system.

The general format of the dump command line is

> java dump *file-name*

where

file-name
: is the name of the file to be dumped. This should generally be the name of the backing file for the file system simulator (e.g., "filesys.dat").

The general format of the dump output is

> *addr hex dec asc*

where

addr
: is the decimal address of the byte,

hex
: is the hexadecimal value of the byte,

dec
: is the decimal value of the byte, and

asc
: is the corresponding ASCII character if the value is between 33 and 127 (decimal).

Each line of dump output corresponds to a single byte in the file. To keep the listing brief, dump only displays non-zero bytes from the input file.

For example

```
java dump filesys.dat | more
```

causes the contents of the file "filesys.dat" to be displayed, one line per byte. The "| more" causes you to be prompted for each page of the output.

The first page of the output should look something like this:

```
0 1 1
5 28 40 (
9 1 1
13 2 2
17 a 10
256 1f 31
512 40 64 @
515 3 3
523 30 48 0
527 ff 255
528 ff 255
529 ff 255
530 ff 255
531 ff 255
532 ff 255
533 ff 255
534 ff 255
535 ff 255
536 ff 255
537 ff 255
538 ff 255
539 ff 255
540 ff 255
541 ff 255
```

You should notice, for example, that the first block (the super block) contains a few numeric values corresponding to the block size (the 1 in the 0 byte means 256), number of blocks, etc. The second block (starting at byte 256) contains a few bits that are set, indicating that the first few blocks are allocated. The third block (starting at 512) contains a few index nodes; the FF/255 values indicate that a direct block is unallocated. A little further down you will see ".", and ".." for the directory entries for the root file system, and other data blocks.

Simulator Configuration File

Each file system simulator program must call Kernel.initialize() before calling any of the other Kernel methods. The initialize() method reads a configuration file ("filesys.conf" is the default), opens the backing file for the file system ("filesys.dat" is the default), and performs other initializations. This section of the user guide describes the various options which may be set in the configuration file.

Configuration File Options

Name	*Description*	*Default Value*
filesystem.root.filename	The name of the file containing the root file system for the simulation.	filesys.dat
filesystem.root.mode	The mode to use when opening the root file system backing file. The mode should either be "rw" for reading and writing, or "r" for read-only access.	rw
process.uid	The numeric user id (uid) to use for the default process context. This should be a number between 0 and 32767.	1
process.gid	The numeric group id (gid) to use for the default process context. This should be a number between 0 and 32767.	1
process.umask	The umask to use for the default process context. This should be an octal number between 000 and 777.	022
process.dir	The working directory in the simulated file system to be used for the default process context. This should be a string that starts with "/".	/root
process.max_open_files	The maximum number of files that may be open at a time by a process. When a process context is created, this many slots are created for possible open files.	10
kernel.max_open_files	The maximum number of files that may be open at one time by all processes in the simulation. When the simulator starts, this many slots are created for possible open files.	20

A Sample Configuration File

In addition to the standard configuration file, "filesys.conf", the distribution also includes a smaller sample configuration file, "sample.conf". This is shown below to illustrate a typical configuration file.

```
!
! my personal filesys configuration file
!
filesystem.root.filename = rayo.dat
filesystem.root.mode = r
process.uid = 1000
process.gid = 1000
process.umask = 002
process.dir = /home/rayo
```

In this particular example, the file system is contained in the backing file "rayo.dat", which is here being opened for read-only access. The working directory for the default process context is "/home/rayo", with the uid, gid, and umask shown.

Specifying an Alternate Configuration File

The default configuration file is named "filesys.conf" and is included in the application distribution. You may modify this file directly to set various options, or you may create your own configuration file and specify the name of this new file when you launch your simulator programs.

If you choose to create your own configuration file, you will need to define a system property "filesys.conf" which contains the name of file. For example, suppose you wanted to run the "ls" program using "my_filesys.conf" as the configuration file. Your java command would look something like this:

```
java -Dfilesys.conf=my_filesys.conf ls /home
```

If there is no value set for the "filesys.conf" system property, then the name "filesys.conf" is used as the default configuration filename.

Writing File System Simulator Programs

Writing programs that use the File System Simulator requires the use of the Kernel class, and may involve the use of the classes Stat and DirectoryEntry. If you're writing ordinary programs that use the standard file system calls, you should not need to reference any other classes.

These three classes are described briefly here. For more information, follow the link for the class to the javadoc for that class.

Kernel
: sets up the simulator environment and defines all the system calls. This class defines: the method initialize(), which is used to initialize the file system simulator; the creat(), open(), read(), write(), close(), and other methods which simulate the work of a file system; and constants like EBADF, S_IFDIR, and O_RDONLY which are used to represent parameter or return values for the system calls. All the methods and fields of Kernel are static; you do not instantiate a Kernel object. For examples, see any of the sample programs (i.e., cat.java, cp.java, ls.java, etc.)

Stat
: is a data structure that represents information about a file or directory. This intends to faithfully represent the Unix stat struct. You may reference fields within a stat object directly (e.g., stat.st_ino), or using JavaBean-style accessor/mutator methods (e.g., stat.getIno() or stat.setIno(). Stat objects are updated by the methods Kernel.stat() and Kernel.fstat(). For examples, see mkdir.java.

DirectoryEntry
: is a data structure that represents a single record in a directory file. This intends to faithfully represent a Unix dirent struct. It contains an index node number and a file name. You may reference the fields directly (e.g., dirent.d_ino), or using JavaBean-style accessor/mutator methods (e.g., dirent.getIno() or dirent.setIno()). However, Java programmers my find it more convenient to use the getName() and setName() (which use String) instead of the field d_name (which is byte[]). DirectoryEntry objects are updated by the method Kernel.readdir(). For examples, see mkdir.java and ls.java.

For more information about Unix system calls and the stat and dirent structs, refer to a Unix system manual. Users of Unix-like systems may find the commands "man -S 2 creat", "man -S 2 open", etc. to be helpful.

All programs that use the File System Simulator should adhere to the following guidelines:

- Invoke the method Kernel.initialize() before any other File System Simulator calls.
- Use Kernel.exit() when you wish to terminate processing in your program.
- Check for errors after each system call (e.g., creat(), open(), read(), write(), etc.). Nearly all the system calls return -1 if an error occurs.
- Use Kernel.perror() to print the message associated with an error.
- Use Kernel.getErrno() to determine which error occurred, if needed. Note that in standard Unix programs you would reference the static process variable "errno".

For examples, take a look at the following sample programs in the distribution:

- cat.java
- cp.java
- ls.java
- mkdir.java
- tee.java

Collectively, these sample programs invoke all of the core methods (system calls) of the file system simulator.

Enhancing the File System Simulator

Adding new features to the File System Simulator is an excellent way to probe your understanding of file system operation, and to investigate new features. Enhancements will almost certainly require changes to the class Kernel, and may necessitate changes to the sample programs described above. This section describes the other classes that implement the functionality of the simulator so that you may understand the intended organization of these components when making a proposed enhancement.

The following are the *internal* classes for the file system simulator:

BitBlock
: is a data structure that views a device block as a sequence of bits. The methods setBit(), resetBit(), and isBitSet() are used to set, reset, or check a bit in the block. This structure is used to implement bitmaps, and is used by the file system simulator to track allocated and free data blocks in the file system. BitBlock extends Block.

Block
: is a data structure that views a device block as a sequence of bytes. The field bytes is an array of byte, and is directly accessible. Included are methods to read() and write() the block to a java.io.RandomAccessFile, which simulate the action of reading or writing a device block.

FileDescriptor
: is a structure and collection of methods that represent an open file. It includes a number of get and set methods for various tidbits of information about the open file, and provides readBlock and writeBlock() methods for reading and writing the blocks of the file.

FileSystem
: is a structure and collection of methods that represent an open (mounted) file system. It includes a few get and set methods for various fields about the file system, but more importantly, includes methods to open() the file behind the file system, to read() and write() blocks of the device, to manage blocks (allocateBlock() and freeBlock()) and to manage inodes (allocateIndexNode()). In general, Kernel methods should call FileSystem methods when they want to read or write data in the file system.

IndexNode
: is a structure and collection of methods for representing an index node. This is meant to reflect the exact structure on disk for an index node. It includes get and set methods for each of the fields in the index node. Also included are read() and write() methods which are used to copy data to and from byte arrays (not disk files).

ProcessContext
: is a structure and collection of methods to represent a process. This is where the simulator stores the uid, gid, umask, dir, and other information for the current process. It includes get and set methods for each of the fields in a process.

SuperBlock
: is a structure and collection of methods for representing the superblock on the disk. In our implementation, the superblock contains information about the block size, number of blocks, offsets to the first block of the free list, inode block, and data block areas of the device. It includes get and set methods for each of the fields in the superblock. Also included are methods to read() and write() the superblock.

Of course, you should look at the code and plan your enhancements carefully.

Please send suggestions, corrections, and comments to Ray Ontko (*rayo@ontko.com*).

Last updated: September 29, 2001