M1 MOSIG
Subproject 5 : Nachos File System

Vincent Danjean, Guillaume Huard, Arnaud Legrand,
Vania Marangozova-Martin, Jean-François Méhaut

2010/2011

File management within a system is an important and complex component. For example, the Unix file system has at least three levels of indirection before reaching data on the disk : a table of open files for each process, the system table of open files, and the inodes table. In addition, searching on disk for information occasionally requires several levels of indirection (large files). Finally, the insertion/withdrawal and search algorithms used can become relatively complex.

In order to implement a fairly simple file system, or one that can at least be learned within a relatively short space of time, the designers of NachOS sacrificed several concepts that can affect the realism of the file system. The main restrictions of this file system are the maximum size of files (37.5 KB), the static allocation of space at file creation, a single-level directory system (no sub- directories), the absence of input/output buffers (no cache memory for the disk) and the lack of robustness. The main asset from these restrictions is that the code for the file system is very small.

The NachOS file system (with its restrictions) is currently functional. You will find an executable version of the NachOS system that makes it possible to test the file system within the directory `nachos/code/filesys` under the name nachos. The commands accepted by this program are `-f` to format the disk, `-cp` to copy a file from the Unix system to the Nachos file system, `-l` to list the content of the directory, and `-r` to delete a file. More extensive information can be obtained in the files `nachos/code/threads/main.cc` and `nachos/code/filesys/fstest.cc`.

The Nachos file system has a single-level directory. When the system starts, Nachos loads the directory's header in memory. The directory contains no more than 10 entries, each containing three pieces of information : an `inUse` boolean that indicates whether the entry is used or not, a string containing the file name, and a pointer to the file header. The file header contains the file size in bytes and sectors, and a number of pointers to data blocks.

Obviously, the file system is stored on-disk. NachOS therefore provides a disk simulator that accepts requests to read from and write to a sector. The disk simulator calculates the time necessary for performing the request, depending on the current status of the disk, and, after the required time, reports the end of input/output via an interrupt. The disk data are stored in a Unix file system. NachOS also provides an interrupt driver that deals with starting the input/output and blocking the thread until the end of input/output is reported (input/output operations are synchronous at the driver's level).

The initial version of the NachOS file system implements a simple and uniform disk management. Therefore, most data structures fit within one sector (headers and data blocks). The free space management is performed by a bits vector stored in sector 0. The directory's header is stored on sector 1.

The file system is implemented within several C++ classes. These classes are as follows :

– The disk (`Disk`)
  This class implements the structure of a disk and provides the sector read and write operations. The size of the disk is initially defined as 128 KB. The code is available in `code/nachos/machine/disk(.h/.cc)`.

- The driver (`SyncDisk`)
  This class defines the structure and routines that implement the disk driver and provide synchronous disk input/output operations. The code for this class is in `nachos/code/filesys/synchdisk (.h/ .cc)`.

- The headers (`FileHdr`)
  This class describes the structure of a file's header. It defines all the information contained in the header, and the operations for manipulating it. The code for this class is in the files : `nachos/code/filesys/filehdr.(.h/.cc)`.

- The directory (`Directory`)
  This class defines the structure of the directory, and the functions that enable it to be manipulated. The code for this class is in the files : `nachos/code/filesys/directory(.h/.cc)`.

- Open files (`OpenFile`)
  This class makes it possible to read and write data in a file. It translates the file read and write operations into operations to read and write sectors on the disk. To achieve this, it uses the file header and the current position within the file. The code for this class is in : `nachos/code/filesys/openfile(.h/.cc)`.

- The file system (`FileSys`)
  This class defines the interface of file system. It provides operations such as file creation, deletion and opening. The code for this class is in : `nachos/code/filesys/filesys(.h/.cc)`.

The interaction between these various objects is fairly simple. When the system initializes, NachOS reads information concerning free space and the directory from the disk. This information is directly available on sectors 0 and 1. When a file is opened, the file system obtains the necessary information from the directory, and returns a pointer to an object of `OpenFile` type. NachOS then uses this pointer to modify the file (read operation, write operation, or positioning within the file). Closing of a file is implicit within NachOS. This occurs when the `OpenFile`-type object is destroyed.

NachOS provides a file system that has plenty of limitations and restrictions. The purpose of this project is to enrich it with new functionalities. Before starting the various parts of this project, you are recommended to read and analyze all the files in the NachOS file system.

Partie I. Implementing a directory tree

NachOS therefore currently provides a single-level directory, and all the files on the disk are in this directory. The implementation of hierarchical directories supposes that a folder can contain files AND subdirectories. New functions must therefore be provided so as to be able to create and delete a directory, change directory, and print the contents of the current directory.

The creation of a directory hierarchy poses new problems. In particular, you have to be able to move about easily within the hierarchy, add a method for remembering the current directory, create special directories for moving about within the hierarchy in the two directions (`.` and `..`) and, finally, the possibility of using path names.

To provide uniform results and facilitate development, the number of entries in a subdirectory is limited to 10 (the `NumDirEntries` constant in the file filesys.cc. Two of these entries are reserved for the directories . and .. . In addition, only names relative to the current directory will be used (no path name). Finally, a directory can only be deleted if it is empty.

Partie II. IOpen Files Table

Currently It is not possible to have multiple open files n NachOS. In fact, to do this, NachOS should have a different object for each open file. Implementing a table for the open files is a simple solution for this problem. The maximum open files will be 10.

Partie III. Concurrent Access to Files

NachOS has threads allowing to execute concurrently kernel functions. However, the current file system does not support concurrent accesses. If there are two threads accessing the same file or the same directory, data may be corrupted or even lost.

A simple solution is to forbid the opening or the removal of an already open file. The concurrency is not to be managed with this solution.

If concurrency is to be managed, you should modify the `Thread` class in order to manage the list of files opened by a thread. The standard solution is to have per thread table, as well as a system table. We will fix the limit for opening files to 10.

Partie IV. Increasing the Maximum File Size

The maximum size of a file is currently `3.75` KB. This limitation is imposed by the size of the headers (one sector) and by the use of a single-level index. Increasing the size to `120` KB will then limit the size of a file to approximately the size of the disk. This augmentation requires the use of at least one level of indirection.

Partie V. Implementing files of variable size

In the current file system, the file size is set when it is created. This makes it possible to allocate all the space necessary for the file as soon as it is created, and to never subsequently modify the header. With files of variable size, the initial file size is always 0. The file grows whenever a write operation is performed at a location that lies beyond the current end of the file. This modification requires continual modifications to the header. Modifications must also be made to the command enabling the contents of a directory to be listed. Indeed, it will now be necessary to display the file size.

Partie VI. Implenting Paths (`path name`)

This si strongly related to part I. To test your path names, you should implement a command interpreter analysing all path names passed as an argument.