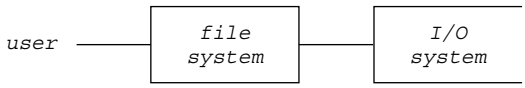
PROJECT 5

File System

1 PROJECT OVERVIEW

This project develops a simple file system using an emulated I/O system. The following diagram shows the basic organization:



The user interacts with the file system using commands, such as *create, open*, or *read* file. The file system views the disk as a linear sequence of logical blocks numbered from 0 to $L - 1$. The I/O system uses a memory array to emulate the disk and presents the logical blocks abstraction to the file system as its interface.

2 THE INPUT/OUTPUT SYSTEM

The physical disk is a two-dimensional structure consisting of cylinders, tracks within cylinders, sectors within tracks, and bytes within sectors. The task of the I/O system is to hide the two-dimensional organization by presenting the disk as a linear sequence of logical blocks, numbered 0 through $L - 1$, where L is the total number of blocks on the physical disk.

We will model the disk as a character array $ldisk[L][B]$, where L is the number of logical blocks and B is the block length, i.e., the number of bytes per block. The task of the I/O system is to accept a logical block number from the file system and to read or write the corresponding block into a memory area specified by the command.

We define the interface between the file system and the I/O system by the following two functions invoked by the system whenever it must read or write a disk block:

- `read_block(int i, char *p);`

  This copies the logical block $ldisk[i]$ into main memory starting at the location specified by the pointer p. The number of characters copied corresponds to the block length, B.

- `write_block(int i, char *p);`

  This copies the number of character corresponding to the block length, B, from main memory starting at the location specified by the pointer p, into the logical block $ldisk[i]$.

In addition, we implement two other functions: one to save the array *ldisk* in a file and the other to restore it. This allows us to preserve the disk contents when not logged in.

3 THE FILE SYSTEM

The file system is built on top of the emulated I/O system described above.

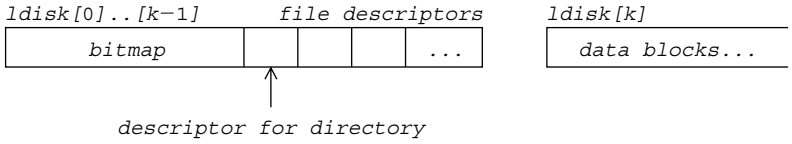
3.1 Interface Between User and File System

The file system must support the following functions: *create, destroy, open, read, write, lseek*, and *directory*.

- *create(symbolic_file_name)*: create a new file with the specified name.
- *destroy(symbolic_file_name)*: destroy the named file.
- *open(symbolic_file_name)*: open the named file for reading and writing; return an index value which is used by subsequent *read, write, lseek*, or *close* operations.
- *close(index)*: close the specified file.
- *read(index, mem_area, count)*: sequentially read a number of bytes from the specified file into main memory. The number of bytes to be read is specified in count and the starting memory address in *mem_area*. The reading starts with the *current position* in the file.
- *write(index, mem_area, count)*: sequentially write a number of bytes from main memory starting at *mem_area* into the specified file. As with the read operation, the number of bytes is given in *count* and the writing begins with the current position in the file.
- *lseek(index, pos)*: move the current position of the file to *pos*, where *pos* is an integer specifying the number of bytes from the beginning of the file. When a file is initially opened, the current position is automatically set to zero. After each read or write operation, it points to the byte immediately following the one that was accessed last. *lseek* permits the position to be explicitly changed without reading or writing the data. Seeking to position 0 implements a reset command, so that the entire file can be reread or rewritten from the beginning.
- *directory*: list the names of all files and their lengths.

3.2 Organization of the File System

The first k blocks of the disk are reserved; they contain the following descriptive information:



The *bitmap* describes which blocks of the disk are free and which are occupied by existing files. Each bit in the bitmap corresponds to one logical disk block. The bitmap is consulted by the file system whenever a file grows as the result of a write operation or when a file is destroyed. (Note that a file never shrinks. The only way to reduce its length is to copy the relevant portion into a new file and to destroy the original file.)

The remaining portion of the first k blocks contains an array of *file descriptors*. The maximum number of descriptors is determined by the block length and the number k. Each descriptor contains the following information:

- file length in bytes;
- an array of disk block numbers that hold the file contents. The length of this array is a system parameter. Set it to a small number, e.g., 3.

3.3 The Directory

There is only one directory file to keep track of all files. This is just like an ordinary file, except it is never explicitly created or destroyed. It corresponds to the very first file descriptor on the disk (see diagram). Initially, when there are no files, it contains length 0 and has no disk blocks allocated. As files are created, it expands.

The directory file is organized as an array of entries. Each entry contains the following information:

- symbolic file name;
- index of file descriptor.

3.4 Creating and Destroying a File

The main tasks performed by the *create* routine are as follows:

- Find a free file descriptor (read in and scan *ldisk [0]* through *ldisk [$k-1$]*)
- Find a free entry in the directory (this is done by rewinding the directory and reading it until a free slot is found; recall that the directory is treated just like any other file). At the same time, verify that the file does not already exists. If it does, return error status.
- Enter the symbolic file name and the descriptor index into the found directory entry
- Return status

To *destroy* a file, the following tasks are performed (assume that a file will not be open when the destroy call is made):

- Find the file descriptor by searching the directory
- Remove the directory entry

- Update the bitmap to reflect the freed blocks
- Free the file descriptor
- Return status

3.5 Opening and Closing a File

There is an *open file table (OFT)* maintained by the file system. This is a fixed length array (declared as part of your file system), where each entry has the following form:

- Read/write buffer
- Current position
- File descriptor index

Whenever a file is opened, an entry in OFT is allocated. When a file a closed, the entry is freed. The first field is a buffer used by read and write operations. The buffer size is the size of one disk block. The second field contains the current byte position within the file for reading/writing; initially it is zero. The third field is an index pointing to the corresponding file descriptor on disk.

The tasks performed by the *open* routine are then as follows:

- Search the directory to find the index of the file descriptor
- Allocate a free OFT entry (if possible)
- Fill in the current position (zero) and the file descriptor index
- Read the first block of the file into the buffer (read-ahead)
- Return the OFT index (or error status)

The essential tasks performed by the *close* routine are as follows:

- Write the buffer to disk
- Update file length in descriptor
- Free the OFT entry
- Return status

3.6 Reading, Writing, and Seeking in a File

When a file is open, it can be *read* and *written*. The read operation performs the following tasks:

1. Compute the position within the read/write buffer that corresponds to the current position within the file (i.e., file length modulo buffer length)
2. Start copying bytes from the buffer into the specified main memory location until one of the following happens:

(a) the desired count or the end of the file is reached; in this case, update current position and return status

(b) the end of the buffer is reached; in this case,

- write the buffer into the appropriate block on disk (if modified),
- read the next sequential block from the disk into the buffer;
- continue with step 2.

Writing into a file is analogous; the data is transferred from main memory into the buffer until the desired byte count is satisfied or the end of the buffer is reached. In the latter case, the buffer is written to disk, the file descriptor and the bitmap are then updated to reflect the new block and the writing continues at the beginning of the buffer. If the file length expands past the last allocated block, a new block must be allocated;

The tasks of the *lseek* operation are as follows:

- If the new position is not within the current data block,
  - write the buffer into the appropriate block on disk
  - read the new data block from disk into the buffer
- Set current position to new position
- Return status

3.7 Listing the Directory

The tasks performed under this command are as follows:

- Read the directory file
- For each entry,
  - find file descriptor
  - print file name and file length

4 THE PRESENTATION SHELL

The functionality of the file system can be tested by developing a set of programs to exercise various functions provided by the file system. To demonstrate your project interactively, develop a presentation shell (similar to the one of the process and resource manager) that accepts commands from your terminal, invokes the corresponding functions of the file system, and displays the results on your terminal. For example, *cr <name>* creates a new file with the specified name; *op <name>* opens the named file for reading and writing and displays an index value on the screen; and *rd <index> <count>* reads the number of bytes specified as *<count>* from the open file *<index>* and displays them on the screen.

It also is very useful to develop two additional support functions: one to save the contents of the array *ldisk[]* in a file, and the other to restore it. This allows the emulated disk to retain its content between login sessions.

5 SUMMARY OF SPECIFIC TASKS

1. Design and implement the emulated I/O system, in particular, the two functions *read_block(int i, char *p)* and *write_block(int i, char *p)*.
2. Design and implement the file system on top of the I/O system. It should support the functions that define the user/file system interface.
3. Define a command language for the presentation shell. Then, design and implement the shell so that you can test and demonstrate the functionality of your file system interactively.
4. Test the file system using a variety of command sequences to explore all aspects of its behavior.

6 IDEAS FOR ADDITIONAL TASKS

1. Extend the directory management to allow the creation and use of tree-structured *subdirectories*. The naming of files and directories can be done using absolute path names or the concept of a current working directory.
2. Implement a multilevel indexing scheme for disk blocks such that a file may grow past the maximum of three blocks assumed in this project.
3. Implement the disk as a four-dimensional array $ldisk[C][T][S][B]$, where C is the number of cylinders, T is the number of tracks per cylinder, S is the number of sectors (physical blocks) per track, and B is the number of bytes per sector. Extend the emulated I/O system such that it accepts the same requests for logical block numbers as before, but it translates these into the actual disk addresses, consisting of cylinder, track, and sector numbers; these are used as indices to access the array *ldisk*.