CS2106 Assignment 3

File Systems

1. Introduction

CONSULTATION: Lab consultations for this assignment will begin 3 April 2017, NOT 27 March. This will give you a chance to begin the assignment before seeking help from the TAs.

In this assignment you will implement a library to read, write and delete files in an Encrypted File System (EFS). You are provided with a basic library (efs.cpp and efs.h), a program to create a partition file on your drive where the new file system will live, and two test programs (testwrite.cpp, testread.cpp) that will create a file in the file system, and read it back to you.

Note: This assignment uses a simple XOR cipher. This is not cryptographically secure and is chosen only because it is simple to implement. If you want to use this assignment for securely storing files, consider using more secure public-key cipher systems.

Also unfortunately due to security restrictions you won't actually be able to mount this file system as an actual drive. Nonetheless it will give you a great understanding of how directories, inodes, etc. work.

2. Deliverables

Package the Answer Book, libefs.h, libefs.cpp, checkin.cpp, checkout.cpp, attrfile.cpp, getattr.cpp and delfile.cpp files into a single ZIP file, with the filename being the matriculation number of the team leader. Upload the ZIP file to the IVLE submission folder by 23:59 hrs on Friday 14 April 2017.

3. Unpacking, Creating and Testing the Executables

Unzip the assignment file to a directory. Change to that directory. A Makefile has been created for you. To create the makefs, testwrite and testread executables, changes to the directory containing Makefile and all the source files, and just type "make".

There is also an "fs.cfg" file holding configuration information for the file system. The file consists of 4 lines:

part.dsk	- Name of the file system partition file
128	- Size of the partition file in megabytes
8192	- Size of each block in bytes
1000	- Maximum number of files allowed

The Encrypted File System lives in a partition file called "part.dsk". To create part.dsk, type:

./makefs fs.cfg

This will create a file called "part.dsk". **NOTE: Doing ./makefs fs.cfg completely erases the partition file. The data cannot be recovered. However this is also a useful way of "reformatting" the partition whenever you want to start afresh.**

To test that your EFS works, use vim or otherwise and create a file called "test.txt". Enter some message (e.g. Hello World, this is EFS!), and save it. Then use:

./testwrite test.txt 2106s2

This will store test.txt in the EFS with password 2106s2. Now delete test.txt using "rm –f test.txt". Use ls to verify that it is gone.

Then to extract test.txt from the EFS, use:

./testread test.txt 2106s2

Now use cat or vim to verify that test.txt is present and intact.

You can attempt extracting test.txt using the wrong password:

./testread test.txt wrongPassword

Again use cat or vim to see that test.txt is now scrambled.

To extract the file again repeat the command:

./testread test.txt 2106s2

You will see the test.txt file restored correctly.

Note you can use "make clean" to purge all old object and executable files.

4. The Encrypted File System Library

The EFS library is in efs.cpp. Definitions and prototypes are in efs.h. This library provides basic low level operations like mounting and unmounting the partition file, manipulating directory entries, free list bitmaps, inodes and reading and writing to blocks.

The tables below show what is in the EFS library.

Constants

Constant	Default Value	Comments
FS_OK	0	File system operation successful. Returned in _result variable.
FS_ERROR	1	Unknown error. Returned in _result.
FS_FULL	99999999	File system full. Returned in _result.
FS_DIR_FULL	88888888	Exceed # of directory entries. Returned in _result.
FS_FILE_NOT_FOUND	77777777	File not in directory. Returned in _Result.
FS_DUPLICATE_FILE	66666666	Attempt to create file with same name as existing file. Returned in _result.
MAX_PASSWD_LEN	32	Maximum length for passwords.
MAX_FNAME_LEN	32	Maximum length for filenames.

Variables

Constant	Default Value	Comments
_result	FS_OK	Returns results of last operation.

Structures

The fields in TFileSystemStruct are handled for you and you can treat them as READ ONLY.

Structure	Fields	Comments
TFileSystemStruct	fsSize	Total size of file system in bytes.
	blockSize	Size of each block in bytes.
	maxFiles	Maximum number of files allowed.
	numBlocks	Number of blocks in file system.
	numInodeEntries	Number of entries in a single inode block.
	bitmapLen	Length of free list bitmap. 1 = free block, 0 = allocated block.
	dirByteIndex	Byte index in partition file to the first directory entry.
	bitmapByteIndex	Byte index in partition file to the first free list bitmap entry.
	inodeByteIndex	Byte index in partition file to the first inode entry.
	dataByteIndex	Byte index in partitional file to the first data block.

TDirectory	filename	Name of file
	length	Length of file in bytes
	attr	File attributes. Bit 0=0 means this directory entry is unused. Bit 1 = 1 means this is a subdirectory entry. Bit 2 = 1 means this is a read/only file.
	inode	Index of inode for this file.

Note: In the operations below, _result variable is always set either to FS_OK or a suitable error code after each operation.

Mounting and Unmounting File Systems

Function	**Parameters**	**Comments**
void mountFS(const char *filename, const char *password)	filename = Name of parititon file. password = Password used to encrypt/decrypt files.	Mounts the specified partition file as a new encrypted file system.
TfileSystemStruct *getFSInfo()	-	Returns a pointer to the file system descriptor. See description of TFileSystemStruct above for fields in this descriptor.
void unmountFS()	-	Updates directory, free list and inode data to the partition file, but DOES NOT flush data buffers!

Directory Manipulation

Function	Parameters	Comments
unsigned int makeDirectoryEntry(const char *filename, unsigned int attr, unsigned long len);	filename = Name of new file to create in directory. attr = File attributes. See TDirectory details above. len = Length of file in bytes.	Returns index to directory entry. You can treat this index as the inode index as well. That is, directory entry 0 always maps to inode 0, entry 1 maps to inode 1, etc. Returns FS_DUPLICATE_FILE if file already exists.
unsigned int updateDirectoryFileLength(const char *filename, unsigned long len)	filename = Name of file to update length. File must exist in directory. len = New length of file.	Returns index to directory entry / inode index. Returns FS_FILE_NOT_FOUND if file does not exist.
unsigned int delDirectoryEntry(const char *filename)	filename = Name of file to remove from directory.	Returns index to directory entry / inode index. Returns FS_FILE_NOT_FOUND if file does not exist.
unsigned int findFile(const char *filename)	filename = Name of file to find.	Returns index to directory entry / inode index. Returns FS_FILE_NOT_FOUND if file does not exist.
unsigned long getInodeForFile(const char *filename)	filename = Name of file to find.	Returns index to directory entry / inode index. Returns FS_FILE_NOT_FOUND if file does not exist.

unsigned long getFileLength(const char *filename)	filename = Get length of this file.	Returns length of file in bytes. Returns FS_FILE_NOT_FOUND if file does not exist.
void setAttr(const char *filename, unsigned int attr)	filename = Name of file to set attribute attr = Attribute for file. See TDirectory for more details.	_result set to FS_FILE_NOT_FOUND if file does not exist.
unsigned int getAttr(const char *filename)	filename = Name of file to get attributes	Returns attributes for specified file
void updateDirectory()	Writes directory back to the partition file.	

Free List Manipulation

Function	**Parameters**	**Comments**
unsigned long findFreeBlock()	-	Returns block number of next free block.
void markBlockBusy(unsigned long blockNum)	blockNum – Block number to mark as busy.	Marks block as busy (allocated).
void markBlockFree(unsigned long blockNum)	blockNum – Block number to mark as free.	Marks block as free (available).
void updateFreeList()	-	Writes free list back to the partition file.

Inode Table Manipulation

Function	Parameters	Comments
unsigned long *makeInodeBuffer()	-	Creates a buffer to store inode data. Use the standard C function free to free the memory after use.
void loadInode(unsigned long *inode, unsigned int inodeNumber)	inode = Pointer to buffer created by makeInodeBuffer inodeNumber = inode to load up.	Specified inode is loaded from the partition file.
void saveInode(unsigned long *inode, unsigned int inodeNumber)	inode = Pointer to buffer created by makeInodeBuffer. inodeNumber = inode to update.	Inode on partition file is updated to contents in the inode buffer.
void setBlockNumInInode(unsigned long *inode, unsigned long byteNumber, unsigned long blockNumber)	inode = Pointer to buffer created by makeInodeBuffer. byteNumber = Byte offset that you want to assign a data block to. blockNumber – Assign data block specified here to this byte offset.	Sets the block number in the inode for a particular byte offset. For example if we had block sizes of 1024 bytes, then setting the block number of byte offset 3372 to 15 would cause inode[3] to be set to 15, since byte offset 3372 corresponds to the 4th entry in the inode.

void returnBlockNumFromInode(unsigned long *inode, unsigned long byteNumber)	inode = Inode buffer created by makeInodeBuffer byteNumber = Byte offset that you want to query	Returns block number corresponding to byte offset, or 0 if block has not yet been allocated for this byte offset. Use findFreeBlock to locate a free block, then setBlockNumInInode to allocate block to this byte offset.

Reading and Writing Blocks

Function	**Parameters**	**Comments**
char *makeDataBuffer()	-	Creates and returns a buffer for storing data. Use the standard C function free to deallocate.
void readBlock(char *buffer, unsigned long blockNum)	buffer = Data buffer allocated by makeDataBuffer. blockNum = Block number to read. Block numbers start from 1, not 0.	Block numbers start from 1, not 0. Data from specified block is read into buffer.
void writeBlock(char *buffer, unsigned long blockNum)	buffer = Data buffer allocated by makeDataBuffer blockNum = Block number to write to. Block numbers start from 1, not 0.	Data in buffer is written to block. Note that block numbers start from 1, not 0.

These tables provide a summary of the functions. Please look at the testwrite.cpp and testread.cpp files to see how they are actually used.

5. Implementing the EFS File Library

The efs.cpp and efs.h provide low level directory, free list, inode and data block management functions. In this assignment you will implement the higher level file manipulation functions in the first part, and write four programs in the second part.

You are provided with the following in libefs.h and libefs.cpp:

File Open Modes

Mode	**Value**	**Remarks**
MODE_NORMAL	0	Open file if it exists, creating entry in open file table and return index. New data will be written to the start of the file.
MODE_CREATE	1	Open file if it exists, otherwise create it. Create entry in open file table and return index. New data will be written to the start of the file.
MODE_READ_ONLY	2	Open file if it exists. Create entry in open file table and return index.
MODE_APPEND	3	Opens file and appends new data to the end. Existing data is retained.

Open File Entry (TOpenFile)

Field	**Value**	**Remarks**
openMode	See File Open Modes for valid values	Mode file is currently opened in.
blockSize	Size of disk block.	Obtain from file system attributes.
inode	Pointer to file's inode.	Same as directory index.
inodeBuffer	Contains copy of file's inode.	Use makeInodeBuffer and loadInode
buffer	Buffer to store block data.	Use makeDataBuffer to allocate, and readBlock/writeBlock.
writePtr	Current offset into data buffer for writing. Goes from 0 to blockSize-1.	When writePtr == blockSize, write block back to disk and allocate new block if necessary.

readPtr	Current offset into data buffer for reading. Goes from 0 to blockSize-1.	When readPtr == blockSize, load next block if any.
filePtr	Current file pointer. Points to next byte in file to read/write.	

See the libefs.h and libefs.cpp files for details on what you need to implement.

6. Implementing EFS Utilities

You have to implement 6 EFS utilities using the libefs library you just created:

Utility Name	**Source File**	**Remarks**
checkin <file> <password>	checkin.cpp	Checks in a file into the EFS using password specified. More advanced version of testwrite using routines in libefs.cpp. Print DUPLICATE FILE error message if file already exists in the EFS.
checkout <file> <password>	checkout.cpp	Checks out a file from the EFS using password specified. More advanced version of testread using routines in libefs.cpp. Print FILE NOT FOUND error message if file is not in EFS>
delfile <file>	delfile.cpp	Removes specified file from EFS. Print FILE NOT FOUND if file is not in the efs.
attrfile <file> <attr>	attrfile.cpp	Use 'R' or 'r' for attr to set file as read oly, and 'W' or 'w' to set as read/write. Print FILE NOT FOUND if file is not in EFS.
getattr <file>	getattr.cpp	Write 'R' to screen if specified file is read only,

		and ‘W’ if it is read write. Print FILE NOT FOUND if file is not in EFS.