File Processing and Application

By:

Mr I. O. Eweoya

File Managers

Introduction

- Various implementations of file management routines are available. Utility software are either integrated with operating system or available as off-the-shelf software.
- A *file manager* or *file browser* is a computer program that provides a user interface to work with file systems. The most common operations are: *Create*, *open*, *edit*, *view*, *print*, *play*, *rename*, *move*, *copy*, *delete*, *attributes*, *properties*, *search/find*, and *permissions*.

File Managers Introduction

- Files are typically displayed in a hierarchy. Some file managers contain features inspired by web browsers, including forward and back navigational buttons.
- Some file managers provide network connectivity such as FTP, NFS, SMB or WebDAV. This is achieved either by allowing the user to browse for a file server, connect to it and access the server's file system like a local file system, or by providing its own full client implementations for file server protocols.

File Managers
Introduction Contd.

- File managers can be orthodox, file list, directory editors, spatial file managers, 3-D file managers, navigational file managers, and web-based file managers.
- However, we concentrate on the last two.

Navigational File Manager

- A *navigational file manager*, also called an ***Explorer type manager***, is a newer type of file manager as commonly found in Microsoft Windows.
- The Windows Explorer is a classic representative of the type, using a "navigational" metaphor to represent file system locations. An exploration of GUI.

Features of Navigational File Manager

- The window displays the location currently being viewed.
- The location being viewed (the current directory) can be changed by the user, by opening folders, pressing a *back button*, typing a location, or using additional pane with the navigation tree representing part or all the file system.
- Icons represent files, programs, and directories.

Features of Navigational File Manager Contd.

- The File navigation operations include Back, Forward, Reload, Drag and Drop, Clipboard Operations. It is possible to view many directories at the same time and perform cut and paste operations between instances.
- The usage of GUI is embraced here. **Examples of Navigational File Manager are: Windows Explorer, Mac OS X Finder, Xtree/ZTreeWin, XYPlorer.**

Web-Based File Manager

- Web-based file managers are typically scripts written in PHP, Perl, Asp or any other server side languages. When installed on a local server or on a remotely hosted server they allow files and folders located there to be managed and edited without the need for FTP access.
- More advanced, and usually commercially distributed, web-based file management scripts allow the administrator of the file manager to configure and secure individual user accounts, each with individual account permissions.

Web-Based File Manager Contd.

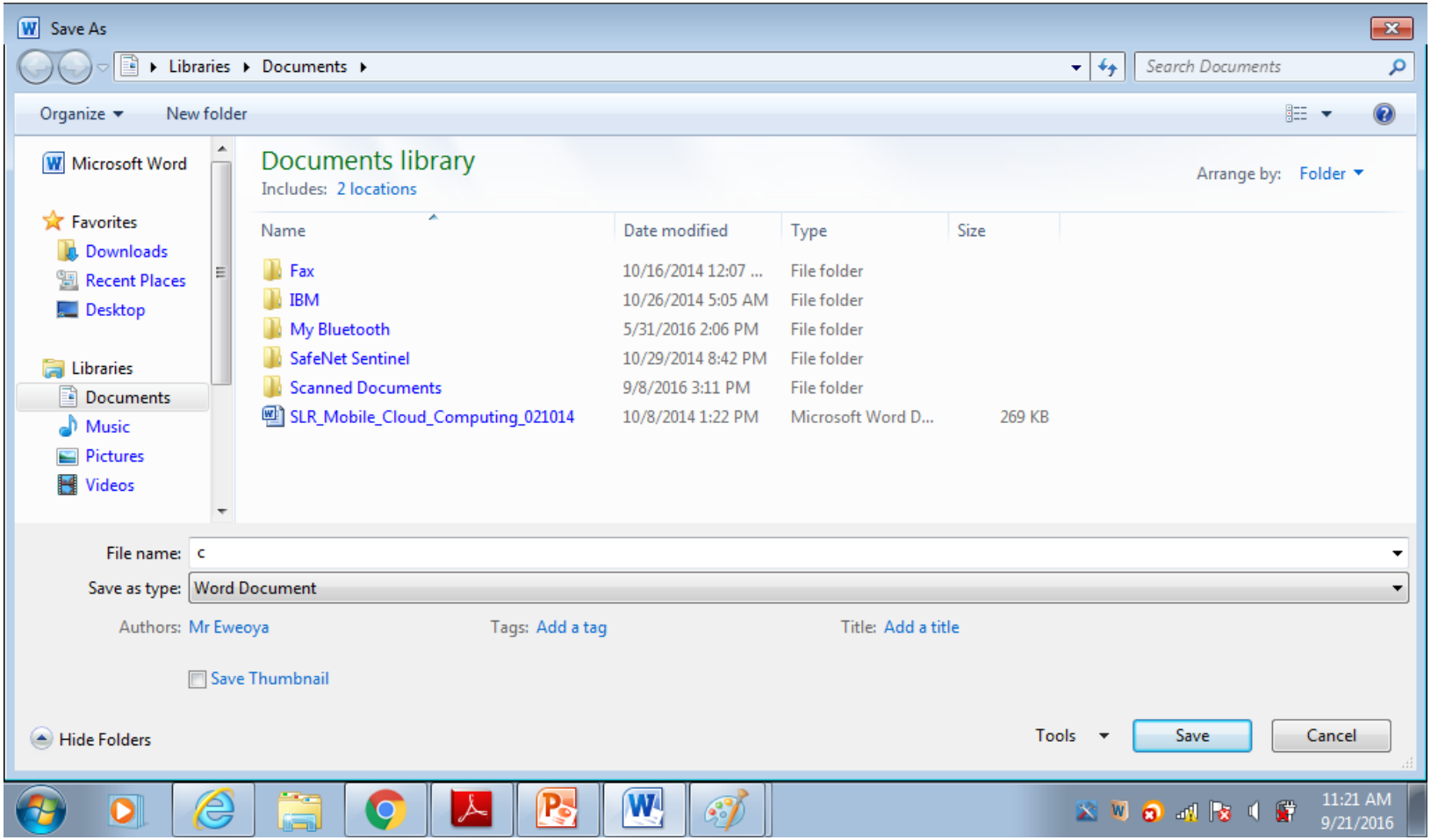
- Authorized users have access to documents stored on the server or in their individual user folders anytime from anywhere via a web browser.
- A web-based file manager can serve as an organization's digital repository. For example, documents, digital media, publishing layouts, and presentations can be stored, managed, and shared between customers, suppliers, remote workers or just internally.

MANAGING FILES IN WINDOWS

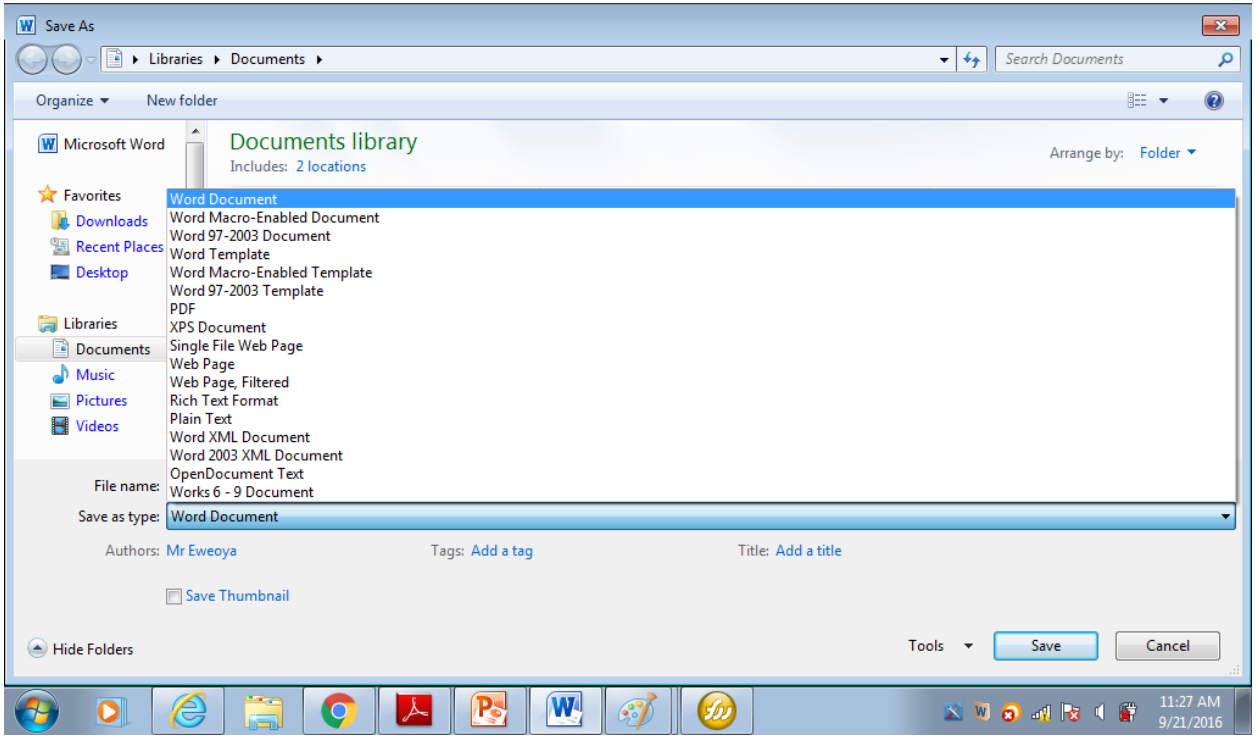
There are basically three ways of managing files in Windows operating systems:

- From within a Program
- By using My Computer
- By using Windows Explorer.

The Save As Dialog Box in Windows



`Save As Type What?



Tips for Management of Electronic Files

- Organize by file types
- One place for all
- Create folders in My Documents
- Nest folders within folders
- Follow the file naming conventions
- Be specific
- File as you go
- Order your files for your convenience
- Cull your files regularly
- Back up your files regularly

FILE SORTING, SEARCHING, AND MERGING

- Computer or electronic files are stored in main memory or secondary storage devices. Retrieval, re-arrangement or manipulation is done on these files for some purposes. Various techniques to achieve the above make the basis of our next discussion.

Sorting

- A **Sorting Algorithm** is a prescribed set of well-defined rules or instructions that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order.
- Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly.

Some Popular Sorting Algorithms

Bubble Sort

It continuously steps through a list, swapping items until they appear in the correct order.

The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them.

It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. While simple, this algorithm is highly inefficient and is rarely used.

Bubble Sort

```
void bubbleSort (int a[ ] , int size)
{
   int i, j, temp;
   for ( i = 0; i < size; i++ )    /* controls passes through the list */
   {
            for ( j = 0; j < size - 1; j++ )   /* performs adjacent comparisons */
            {
                     if ( a[ j ] > a[ j+1 ] )   /* determines if a swap should occur */
                     {
                              temp = a[ j ];      /* swap is performed */
                              a[ j ] = a[ j + 1 ];
                              a[ j+1 ] = temp;
                     }
            }
     }
}
```

Insertion Sort

This is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often used as part of more sophisticated algorithms.

It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list.

In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort is a variant of insertion sort that is more efficient.

Shell Sort

An improvement upon bubble sort and insertion sort by moving out of order elements more than one position at a time.

One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements.

Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second.

It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list.

Of the algorithms described here, this is the first that scales well to very large list.

Heap Sort

Heap sort is a more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a *heap*, a special type of *binary tree*.

Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is re-arranged so the largest element remaining moves to the root.

Heapsort Picture

FIGURE 10.8
Example of a Heap with Largest Value in Root

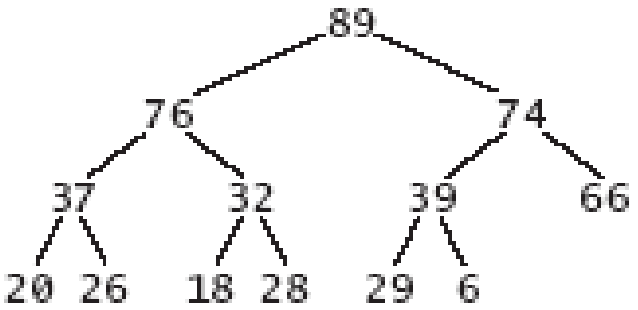


FIGURE 10.9
Heap After Removal of Largest Item

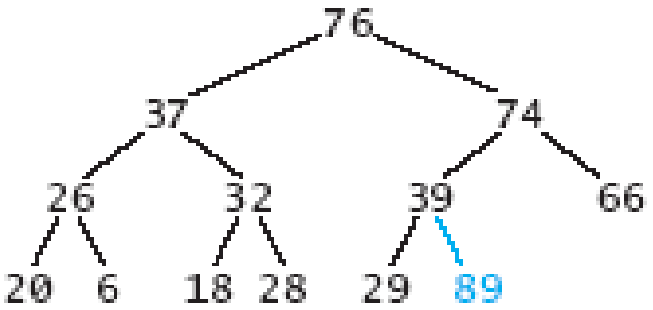


FIGURE 10.10
Heap After Removal of Two Largest Items

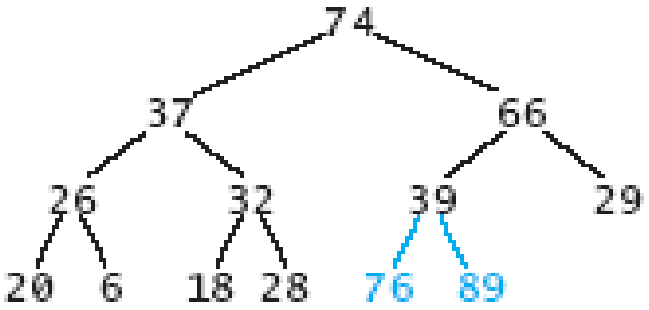
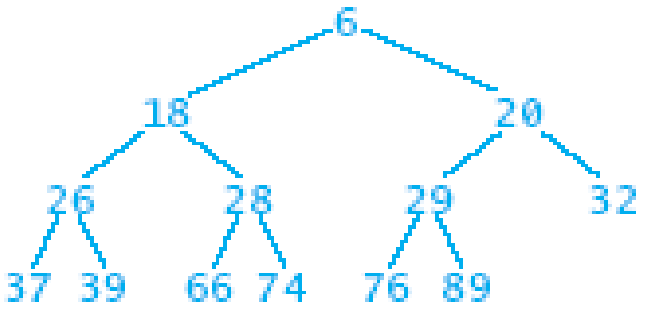


FIGURE 10.11
Heap After Removal of All Its Items



Quick Sort

Quick sort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array, we choose an element, called a *pivot*, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in linear time and in-place.

We then, recursively sort the lesser and greater sub-lists. Efficient implementations of quick sort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice.

Quick Sort (2)

Because of its modest space usage, quick sort is one of the most popular sorting algorithms, available in many standard libraries.

The most complex issue in quick sort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower performance, but if at each step we choose the *median* as the pivot then it works with better performance.

Bucket Sort

Bucket sort is a sorting algorithm that works by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

Thus this is most effective on data whose values are limited (e.g. a sort of a million integers ranging from 1 to 1000). A variation of this method called the single buffered count sort is faster than quick sort and takes about the same time to run on any set of data.

Radix Sort

Radix sort is an algorithm that sorts a list of fixed-size numbers of length k by treating them as bit strings. We first sort the list by the least significant bit while preserving their relative order using a stable sort.

Then we sort them by the next bit, and so on from right to left, and the list will end up sorted. Most often, the counting sort algorithm is used to accomplish the bitwise sorting, since the number of values a bit can have is minimal - only ‘1’ or ‘0’.

Comparison of Sort Algorithms

TABLE 10.4
Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Searching Techniques

A *search algorithm* is simply an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions.

The set of all possible solutions to a problem is called the ***search space.***

Uninformed Vs Informed Search

Uninformed Algorithms use *Brute-force search*, also called a naïve search, it uses the simplest method of searching through the search space one by one for the expected result.

Informed Search Algorithms use *heuristic functions* to apply knowledge about the structure of the search space to try to reduce the amount of time spent searching.

Uninformed Search –Any Drawback?

Uninformed Search - Not specific to the nature of a problem, can be implemented in general, and then the same implementation can be used in a wide range of problems due to abstraction.

The drawback is that, in actual practice, many search spaces are extremely large, and an uninformed search (especially of tree or graph storage structures) will take a reasonable amount of time for even relatively small examples.

(a)List Search
Linear Search

The simplest of such an algorithm is *linear search*, which examines each element of the list as they are encountered. It is expensive in running time compared to many other algorithms.

It can be used directly on any unprocessed list, regardless of history, which is sometimes useful. A more sophisticated list search algorithm is the binary search.

Binary Search

This is significantly better than linear search for large lists, but is sometimes useful for surprisingly small ones given its increase in speed. Also, it requires the list be sorted before searching and generally that the list be randomly accessible.

This may force lengthy reads of mass storage before a binary search can be started. Interpolation search is better than binary search for large sorted lists with ‘fairly even’ key.

Hash Tables

Hash tables can also be used for list searches. They run in constant time in the average case, but have terrible worst-case time. In addition, more space and time is required to set up such tables.

Another search based on specialized data structures uses self-balancing binary search trees. Such tree searches can be seen as extending the ideas of binary search to allow fast insertion and removal in the data structures.

(b)Tree Search

Tree search algorithms are likely the most used searching techniques for structured data. They examine trees of nodes, whether the tree is explicit or implicit (i.e., generated the search algorithm's execution).

The basic principle is that a node is taken from a data structure, its successors examined and added to the data structure. By manipulating the data structure, the tree can be explored in different orders.

Tree Search Contd.

For instance, level by level (i.e., breadth-first search) or reaching a leaf node first and backtracking (i.e., depth-first search), etc.

Other examples of tree-searches include:

- Iterative-deepening search
- Depth-limited search,
- Bidirectional search, and
- Uniform-cost search.

(c)Graph Search

Many of the problems in graph theory can be solved using graph traversal algorithms, such as *Dijkstra's algorithm*, *Kruskal's algorithm*,

the *nearest neighbour algorithm*, and *Prim's algorithm*.

These can be seen as extensions of the tree-search algorithms.

Informed Search
a) Adversarial Search

In an informed search, a heuristic that is specific to the problem is used as a guide. A good heuristic will make an informed search dramatically and out-perform any uninformed search.

Game-playing computer programs, as well as other forms of artificial intelligence like machine planning often use search algorithms like the mini-max algorithm, search tree pruning, and

alpha-beta pruning.

Constraint Satisfaction

This is a type of search which solves constraint satisfaction problems rather than looking for a data value. The solution sought is a set of values assigned to a set of variables.

Because the variables can be processed in any order, the usual tree search algorithms are not suitable.

Constraint Satisfaction

Methods of solving constraint problems include combinatorial search and backtracking, both of which take advantage of the freedom associated with constraint problems.

Common tricks or techniques involved in backtracking are 'constraint propagation', which is a general form of 'forward checking'. Other local search algorithms, such as generic algorithm, which minimize the conflicts may also be practical.

Merge Algorithm

Merge algorithms are a family of algorithms that run sequentially over multiple sorted lists, typically producing more sorted lists as output.

This is well-suited for machines with tape drives. Its usage has declined due to large random access memories, and many applications of merge algorithms have faster alternatives when a random-access memory is available.

Merging Language Support

The C++'s Standard Template Library has the function std::merge, which merges two sorted ranges of iterators, and std::inplace_merge, which merges two consecutive sorted ranges *in-place*.

In addition, the std::list (linked list) class has its own merge method which merges another list into itself. The type of the elements merged must support the less-than (<) operator, or it must be provided with a custom comparator.

PHP has the array_merge() and array_merge_recursive() functions.

File Handling in High Level Languages

Programming languages have facilities embedded in them to read data values from a file stored in the computer memory or any storage device.

The syntax differs from one programming language to another.

The operating system has efficient ways of managing various files generated by any programming language

C Language File Input/Output

The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library header <stdio.h>.

The I/O functionality of C is fairly low-level by modern standards; C abstracts all file operations into operations on streams of bytes, which may be "*input streams*" or "*output streams*".

C Language File Input/Output Contd.

The stream model of file I/O was popularized by the Unix operating system, which was developed concurrently with the C programming language itself.

A large number of modern operating systems have inherited streams from Unix, and many languages in the C programming language family have inherited C's file I/O interface with few if any changes (for example, PHP). The C++ standard library reflects the “stream” concept in its syntax.

Opening a File Using Fopen

A file is opened using **fopen**, which returns an I/O stream attached to the specified file or other device from which reading and writing can be done. If the function fails, it returns a null pointer.

The related C library function **freopen** performs the same operation after first closing any

open stream associated with its parameters.

Opening a File Using Fopen Contd.

- FILE *fopen(const char *path, const char *mode);

- FILE *freopen(const char *path, const char *mode, FILE *fp);

Closing a Stream Using fclose

The fclose function takes one argument: a pointer to the FILE structure of the stream to close. int fclose(FILE *fp);

The function returns zero on success, or EOF on failure. The program **on the next slide** opens a file named sample.txt, writes a string of characters to the file, then closes it.

Opening, writing, and closing a file The file is named sample.txt

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(void)
{
FILE *fp;
size_t count;
const char *str = "hello\n";
fp = fopen("sample.txt", "w");
if(fp == NULL) {
perror("failed to open sample.txt");
return EXIT_FAILURE;
}
count = fwrite(str, 1, strlen(str), fp);
printf("Wrote %zu bytes. fclose(fp) %s.\n",
count, fclose(fp) == 0 ? "succeeded" :
"failed");
fclose(fp);//close de file
return EXIT_SUCCESS;
}
```

Classwork

- Write a C program that opens a binary file called *myfile*, reads five bytes from it, and then closes the file.

Solution to the Classwork

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
char buffer[5] = {0}; /* initialized to
zeroes */
int i, rc;
FILE *fp = fopen("myfile", "rb");
if (fp == NULL) {
perror("Failed to open file \"myfile\"");
return EXIT_FAILURE;
}
for (i= 0;(rc = getc(fp))!= EOF && i < 5;
buffer[i++]= rc) ;
fclose(fp);
if (i == 5) {
puts("The bytes read were...");
printf("%x %x %x %x %x\n", buffer[0],
buffer[1], buffer[2], buffer[3], buffer[4]);
} else
fputs("There was an error reading the
file.\n", stderr);
return EXIT_SUCCESS;
}
```

Text File Operations in C#

C-Sharp provides a File class which is used in manipulating text files.

The File class is within the System namespace. Also we can use the StreamReader and StreamWriter classes, which are within the System.IO, namespace for reading from and writing to a text file.

Now, we see examples of creating a text file, reading contents of a text file and appending lines to a text file.

Creating a Text File

For creating text file we use the CreateText Method of the File Class.

The CreateText Method takes in the path of the file to be created as an argument. It creates a file in the specified path and returns a StreamWriter object which can be used to write contents to the file. An example is presented on the next slide.

Creation of a Text File in C#

```
public class FileClass
{
public static void Main()
{
WriteToFile();
}
static void WriteToFile()
{
StreamWriter SW;
SW=File.CreateText("c:\\MyTextFile.txt");
SW.WriteLine("God is greatest of them all");
SW.WriteLine("This is second line");
SW.Close();
Console.WriteLine(“File Created
SuccessFully”);
}
}
```

Reading Contents of a Text File

For reading the contents of a text file we use the OpenText Method of the File class. The OpenText Method takes in the path of the file to be opened as an argument.

It opens the specified file and returns a StreamReader object which can be used to read the contents of the file. For example:

Reading Contents of a Text File using C#

```
public class FileClass
{
public static void Main()
{
ReadFromFile("c:\\MyTextFile.txt");
}
static void ReadFromFile(string filename)
{
StreamReader SR;
string S;
SR=File.OpenText(filename);
S=SR.ReadLine();
while(S!=null)
{
Console.WriteLine(S);
S=SR.ReadLine();
}
SR.Close();
}
}
```

Appending Content to a Text File

For appending content to a text file we use the AppendText Method of the File class. The AppendText method takes in the path of the file to which the contents to be appended as an argument.

It opens the file in the specified path and returns a StreamWriter object which can be used to append contents to the file. The next slide is the code.

Appending Content to a Text File with C#

```
public class FileClass
{
public static void Main()
{
AppendToFile();
}
static void AppendToFile()
{
StreamWriter SW;
SW=File.AppendText("C:\\MyTextFile.txt");
SW.WriteLine("This Line Is Appended");
SW.Close();
Console.WriteLine("Text Appended
Successfully");
}
}
```

References

Davis, W. S. & Rajkumar, T.M. (2001). Operating Systems: A Systematic View (5th Edition). New Jersey: Addison-Wesley.

Deitel, H. M. (2004). Operating Systems, (3rd Edition). New Jersey: Prentice Hall.

Encarta (2007) http://en.www.wikipedia.com.

Stallings, Williams (2004). Operating Systems; Internal and Design Principles, (5th Edition). New Jersey: Prentice Hall.

Tanenbaum, A. S. (2006). Modern Operating System, (3rd Edition). New Jersey: Prentice Hall.

Williams, B. K. (1999). Using Information Technology: A Practical Information to Computers and Communications. (3rd Edition). Boston: McGraw Hill.

Ismaila O. Mudasiru, (2011), File Processing and Management, Course Guide, NOUN.