VUBWN Contact Us for LMS Handling
NASIR ABBAS

CS609 Handouts BY VUBWN

VUBWN Contact Us for LMS Handling NASIR ABBAS

Table of Contents

Lesson 1

Topic 1: Windows Operating System

Software can be categorized into two main types: **System Software** and **Application Software**. System Software is concerned to the system or its resources while Application Software is related to some application. Operating System is one of the good examples of System software that acts as a manager for the System's resources. It arbitrates and schedules the resources among the processes to avoid any kind of conflicts.

There are varieties of Operating Systems available in the market but Windows is one of the important Operating Systems developed by Microsoft. It is widely used inside the PCs, Laptops, enterprise servers, handheld devices and cell phones etc.

Core Features of Windows Operating System

1. **Memory Management**. Memory Management is the vital or key feature of Windows Operating System. The OS is responsible for managing primary as well as secondary memory. Virtual address space on secondary memory is also managed by OS to transfer data between primary and secondary storage and accommodate a large process in small memory space. All kinds of services and related data structures are supported by Operating Systems to efficiently manage the Virtual memory space.
2. **File systems**. Windows OS manages all kinds of files and folders on disk in hierarchical form. Large file naming space of 255 characters is supported by the OS. Number of APIs are provided by the OS to access the files in both Sequential and Random mode.
3. **Processors**. Windows Operating System provides support to multiprocessors and multi cores systems. It also arbitrates among the cores or processors to efficiently divide and allocate computational tasks to them.
4. **Resource naming and location**. In Windows OS, the resources like processes or devices are treated as objects and each object is assigned a unique name to identify, locate and access it.
5. **Multitasking**. Windows OS supports multitasking. It manages processes, threads, and other independent units, and their asynchronous execution. Tasks can be preempted and scheduled according to dynamically calculated priorities.
6. **Communication and Synchronization**. The Windows OS provides constructs to manage inter-process communication and synchronization within a computer or networked computers.

7. **Security and Protection**. The Windows OS has a strong security mechanism to protect resources from illegal and accidental access. A user cannot access other user data without assigning privileges.

Lesson 2

Topic 2: Windows Evolution

Windows exist in several versions. New versions of Windows are introduced from time to time. Actually, certain new APIs are included in the new version to improve or extend its functionalities. The following major themes or features are considered while developing a new version.

- **Scalability**. A new version of Windows OS runs on different platforms including PCs, enterprise servers, multiprocessing systems, mobiles and systems having large memory space.
- **Performance**. A newer version of Windows certainly improves performance compared to previous versions.
- **Integration**. A newer version must integrate with new technologies like web services, .NET technologies, multimedia etc.
- **Ease of use**. Certain new APIs and improved GI in the new version can ensure ease of use.
- **Enhanced API**. Introducing new APIs or enhancements in existing APIs should be the main theme of the new version.

Disk Operating System (DOS)

In the 1980s, Microsoft Disk Operating System was used inside the IBM PCs incorporating Intel Processor. It was a text based and command line operating system. It was a single user OS. Its filing system was based on FAT and was able to access up to the maximum of 4GB files.

Windows History

Keeping the demand of graphical user interface, Microsoft developed its first version Windows 3.1. In this version, DOS Kernel and FAT based file systems were used.

After that in the 1990s, certain new versions of Windows named Windows 95, 97 and 98 were introduced supporting the 32-bit architecture of Intel's processors.

Later on, Windows NT versions were introduced supporting a file system based on new technology called NTFS. Its security and file system were better than the previous versions.

Windows Server 2008 OS was developed for professional use to manage enterprise and server applications. Support for multi-core technology and 64-bit applications was provided in this OS. Other Windows versions supporting 32-bit, 64-bit architecture, multi-core and multiprocessing were also introduced including Windows XP, Windows Vista, Windows 7, 8 and Windows 10.

Lesson 3

Topic 3: Windows Market Role

Several competitors of Windows OS like UNIX, Linux etc. exist in the market; however, Windows has its own unique status in the market. It has several significant advantages over other operating systems.

Above 90% PCs are based on Intel's processors and Windows is the most appropriate OS for Intel PCs. In the world of desktop, the most dominant OS is the Microsoft Windows which enjoys a market share of above 80%. Windows is not confined to the desktop, it also has support for diverse platforms including multi-core, multiprocessing, servers and mobiles etc.

Due to its dominance role, certain applications and software development tools are available in the market that can easily integrate with Windows OS and can develop windows applications ranging from small scale to enterprise level.

One of the key features of Windows OS is its rich GUI that makes its use very convenient. This interface can be easily customized according to the local setup. The size, color and visibility of graphical interface objects can also be changed by the user.

Compared to other operating systems, certain modern features exist in Windows due to which most of the developers develop their applications for Windows targeting the huge market of Windows.

Lesson 4

Topic 4: Windows, Standards and Open Systems

Open-Source Software is a software that is publicly available with its source code to use, modify and distribute with original rights. It is developed by the community rather than a single company or vendor. In contrast, proprietary software is copyrighted and only available to use under a license.

Windows Operating system is a proprietary and copyrighted software of Microsoft Corporation. It is provided for use only under a License agreement. Without purchasing a license, its use is illegal and is an act of copyright infringement.

Being a closed system, Windows has the following strengths.

As Windows components are provided and updated only by a single vendor, its implementation remains uniform throughout the world. Further, extensions in Window components or APIs are only vendor-specific and so no non-standard extension is possible except for platform differences.

Windows also support various types of hardware platforms like open systems.

Interoperability of Windows: Windows provide interoperability with non-window components.

- Windows OS provides support to the Standard C and C++ libraries. We can install and use any C compilers on Windows systems.
- Socket is a resource that is required for interface when two computers are interconnected to each other. Windows also supports sockets to communicate among devices having different computer architectures and access to TCP/IP and other networking protocols.
- It also supports the Remote Procedure Calls (RPCs) architecture to call the remote functions in distributed client-server-based applications.
- Windows also supports the X Windows system which is open source, cross platform software providing GUI in a distributive network environment.

Lesson 5

Topic 5: Windows Principles

- In Windows OS, all the system resources including processes, threads, memory, pipes, DLL etc. are represented by objects which are identified and referenced by a handle. These objects cannot be directly accessed. In case, if any application approaches to access these objects directly, Windows throws an appropriate exception. The only way to access and operate on these objects is a set of APIs provided by Windows. Several APIs can be related to a single object to manipulate it differently.
- A long list of parameters is associated with each API where each parameter has its own significance but only few parameters are specified for a specific operation.
- To perform the task of multitasking and multi-threading efficiently, Windows provides a number of synchronization constructs to arbitrate among the resources.
- The names of Windows APIs are long and descriptive for its proper and convenient use.
- Some pre-defined data types required for Windows APIs are:
    - BOOL (for storing a single logical value)
    - HANDLE (a handle for object)
    - LPTSTR (a string pointer)
    - DWORD (32-bit unsigned integer)
- Windows Data types avoid the pointer operator (*).
- Some lowercase prefix letters with variable names are used to identify the type of variable. This notation is called Hungarian notation. For example, in the variable name **lpszFilename**, 'lpsz' is Hungarian notation representing a long pointer to zero terminated string.
- **windows.h** is a header file including all the APIs prototypes and data types

Lesson 6

Topic 6: 32-Bit and 64-Bit Source Code Portability

Windows keeps two versions of each API, one for 32-bit and other for 64-bit. A 32-bit code can be run on 64-bit hardware but will be unable to exploit some features of 64-bit like accessing large disk space or using large pointer or 64-bit operation.

Latest versions of Windows support both 32 and 64-bit architectures by keeping two versions of each API, one for 32-bit and other for 64-bit.

Interoperability of 32 and 64-bit:

A single source code can be built for 32-bit as well as 64-bit versions. To decide whether executable code of 32 or 64-bit is generated by the compiler at runtime, it depends on its settings or configuration. Further, to decide which version of API is used, it is also based on the compiler's configuration.

A 32-bit code can run on 64-bit hardware successfully but will be unable to use some features of 64-bit like large disk space, large pointer etc.

A source code developed for 64-bit architecture cannot easily run on a 32-bit machine. For this purpose, re-compilation of the program is required and suitable configuration is made in the compiler to generate a 32-bit executable code.

Lesson 7

Topic 7: When to use Standard C Library for File Operations

Windows provides a set of built-in APIs to perform I/O operations. A related API with specific parameters is invoked for the concerned resource and I/O operation is performed.

Similarly, certain C/C++ standard functions are available to perform I/O operations. For example, fopen(), fclose(), fread(), fwrite() etc. are C functions that can be used to perform I/O operations related to files.

Differences between Windows APIs and Standard Functions

Standard C functions can be used inside the source code to run on Windows platform because Windows has system calls at low level to support C/C++ functions for I/O operations.

If it is required to run a program on cross-platform, then it is preferred to use the standard C/C++ library functions inside the source code. However, in this case, the advanced Windows features like locking, synchronization, asynchronous I/O, inter process communication etc. cannot be achieved.

In case, if portability is not focused and required to avail the advanced Windows features, then it is preferred to use the Windows APIs.

Lesson 8

Topic 8: A Simple File Copy Program Using Standard C Library

```
// cpC.  Basic File Copy Program:  C Library Implementation

// Copy File1 to File2

#include<stdio.h>

#include<errno.h>

#define BUF_SIZE 256

int main(int argc, char *argv[])      {

FILE *inFile, *outFile;

           char rec[BUF_SIZE];

           size_t  bytesIn, bytesOut;

           if(argc!=3)     {

              printf("Usage: cp file1 file2\n");

              return 1;      }

           inFile=fopen(argv[1], "rb");
```

```
            if(inFile==NULL)  {
               perror(argv[1]);
               return 2;
                              }
            outFile=fopen(argv[2], “wb”);
            if(outFile==NULL)   {
               perror(argv[2]);
               return 3;          }
            /* Process the input file a record at a time */
            while((bytesIn = fread(rec, 1, BUF_SIZE, inFile)) > 0)
{
               bytesOut=fwrite(rec, 1, bytesIn, outFile);
               if(bytesOut != bytesIn)   {
                 perror(“Fatal write error”);
                 return 4;          }
            }
            fclose(inFile);
            fclose(outFile);
            return 0;
}
```

This program is used to copy one file to another using C standard functions. In this program, a buffer of size 256 bytes is used in which the chunks of file are copied one by one.

The source file is opened in read binary mode and the destination file in write binary mode using the C **fopen()** function.

If both are successfully opened, then a file is read inside a loop chunk by chunk using **fread()** function and written onto the destination file using **fwrite()** function. After a few iterations, the file will be written to the destination file and both files are closed.

Lesson 9

Topic 9: A Simple File Copy Program Using Windows APIs

Program 1 - 2

```
// cpC.  Basic File Copy Program:  Windows Implementation
// Copy File1 to File2
#include<stdio.h>
#include<windows.h>
#include<stringapiset.h>
#define BUF_SIZE 16384
int main(int argc, char *argv[])      {
HANDLE  hIn, hOut;
DWORD  nIn, nOut;
           CHAR buffer[BUF_SIZE];
           LPWSTR lpwszFile1, lpwszFile2;
           INT iLen1, iLen2;
```

```
if(argc !=3)   {

               fprintf(stderr, “Usage: cp file1 file2\n”);

               return 1;    }

           lpwszFile1 = (LPTSTR)malloc(510);

           lpwszFile2 = (LPTSTR)malloc(510);

           iLen1 = MultiByteToWideChar(CP_ACP, 0, argv[1], -1,
lpwszFile1, 510);

           iLen2 = MultiByteToWideChar(CP_ACP, 0, argv[2], -1,
lpwszFile2, 510);

           hIn=CreateFile(lpwszFile1, GENERIC_READ, FILE_SHARE_READ,
NULL,

                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

           if (hIn==INVALID_HANDLE_VALUE)  {

                fprintf(stderr, “Cannot open input file. Error:
%x\n”, GetLastError());

                return 2;       }


hOut=CreateFile(lpwszFile2, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,

                     FILE_ATTRIBUTE_NORMAL, NULL);

           if (hOut== INVALID_HANDLE_VALUE)   {

                fprintf(stderr, “cannot open output file. Error:
%x\n”, GetLastError());

                CloseHandle(hIn);

                 return 3;

           }
```

```
            while((ReadFile(hIn, buffer, BUF_SIZE, &nIn, NULL) && nIn >
0)  {

               WriteFile(hOut, buffer, nIn, &nOut, NULL);

               if (nIn != nOut)   {

                  fprintf("Fatal write error: %x\n", GetLastError());

                  CloseHandle(hIn);

                  CloseHandle(hOut);

                  return 4;

                  }

            }

            CloseHandle(hIn);

            CloseHandle(hOut);

            return 0;

}
```

In this program, Windows APIs are used instead of C standard functions. In the main program, the words in capital letters like HANDLE, DWORD, CHAR, LPWSTR and INT are Windows data types. As the file paths given in command line parameters are in ASCII format, the parameters will be first converted to Unicode using the **MultiByteToWideChar()** function.

After that, a file is opened for reading purposes and its handle is stored in hIn. Similarly, another file is created for writing purposes and its handle is stored in hOut.

If files are successfully opened, then the source file is read in a loop and written to the destination file. At the end both the handles for files are closed.

Lesson 10

Topic 10: A Simple File Copy Program Using Windows Convenience Function

Numerous Windows functions are used to perform various tasks at low level. However, Windows has a set of **Convenience functions** that combine several functions to perform a common task. In most cases, these functions improve the performance because several tasks are performed by a single function.

For example, **CopyFile()** is a convenience function that replaces the algorithms used for creating, opening, reading and writing one file to another.

Program 1 - 3

```
/* cpC.  Basic File Copy Program:  Windows Implementation
                using convenience function CopyFile() */
// Copy File1 to File2


#include<stdio.h>
#include<windows.h>
#define BUF_SIZE 256
LPWSTR lpwszFile1, lpwszFile2;
INT iLen1, iLen2;


int main(int argc, char *argv[])      {
if(argc !=3)   {
              fprintf(stderr, "Usage: cp file1 file2\n");
```

```
            return 1;

                    }

          lpwszFile1 = (LPTSTR)malloc(510);

          lpwszFile2 = (LPTSTR)malloc(510);

          iLen1 = MultiByteToWideChar(CP_ACP, 0, argv[1], -1,
lpwszFile1, 510);

          iLen2 = MultiByteToWideChar(CP_ACP, 0, argv[2], -1,
lpwszFile2, 510);



if (!CopyFile(lpwszFile1, lpwszFile2, FALSE)

               {

                                   fprintf(stderr, "CopyFile Error:
%x\n", GetLastError());

                                   return 2;

               }

               return 0;

}
```

Lesson 11

Lecture 11: Windows File System

Windows supports various file systems.

- **NT File System**

NTFS is an important file system supported by Windows, its main features are:

- **Security**: One user cannot access other user data without privileges.
- **Fault tolerance** (if a portion of disk corrupts, it works because different copies of information are maintained in this files system).
- **Encryption** (data encrypted/decrypted, provide security)
- **Compression** (Space capacity increased due to data compression)
- Supports very huge file size

- **File Allocation Tables**
  - FAT12, FAT16, FAT32 (Start version of File Systems, also supported by Windows)
- Compact Disk File System (CDFS)
  - Support CD
- Supports Universal Disk Format (UDF) & Live File System (LFS) also

Lesson 12

Lecture 12: File Naming Conventions

- Windows OS supports a number of File Systems. Each file system has its own mechanism for naming files.

Certain Limitations for File Naming

- Letters like A, B, C etc. are used to represent Drive, Network Drives are represented by higher letters like N, K, L etc.
- Double slash (\\) in the start of path represents remote resource
- Forward slash (\ ) or backslash (/) is used as a path name separator

- The first 31 ASCII characters (control characters) cannot be used in file names
- Special symbols like \, /, colon (:), pipe (|) etc. cannot be used in filenames
- File and directory names are case insensitive
- File name and extensions are separated by (.)
- Max size for file name is 255 and for path is 260 characters.
- File extension takes 2 – 4 bytes
- Single dot (**.**) represents current directory while double dot (**..**) represents one step back (up) directory

Lesson 13

Lecture 13: Creating and Opening Files (Using Windows API)

The CreateFile() API

CreateFile() API with a list of parameters is used to open or create a new file. Its return type is HANDLE to an open file object in case of successful opening or creation. The parameters are:

- **lpFileName**: It is a string pointer that points to a filename to be opened or created.

- **dwDesiredAccess**: It is a 32-bit double word which specifies the GENERIC_READ and WRITE access.

- **dwShareMode**: This mode specifies how the file is shared?
- 0 signifies that file will not be shared
- FILE_SHARE_READ allows the file to be shared for concurrent read
- FILE_SHARE_WRITE allows the file to be shared for writing.

- **lpSecurityAttributes**: It points to a security attributes structure.

- **dwCreate**: signifies whether to create a new file or overwrite an existing one.
    - CREATE_NEW: Creates a new file. If the file already exists, then fail.
    - CREATE_ALWAYS: Creates a new file or overwrites an existing one.
    - OPEN_EXISTING: open an existing file or fail if the file does not exist
    - OPEN_ALWAYS: open an existing file, if it does not exist, then create it.

- **dwFlagsAndAttributes**: It signifies attributes of the newly created file.

- **hTemplateFile**: It is the Handle of an open file that specifies attributes to apply to a newly created file.

Lesson 14

Lecture 14: Reading/Writing a File

The **ReadFile()** API is used to read data from file to buffer.

Syntax:

BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);

- If the file is not opened in concurrent mode, then ReadFile() starts reading from the current position.
- If the current location is End of File, then no Errors occur and *lpNumberOfBytesRead is set to zero
- The function returns FALSE if it fails in case any of the parameter is invalid

Parameters

- HANDLE hFile is the file handle
- LPVOID lpBuffer is the address of the array that stores the data read from the file.
- DWORD nNumberOfBytesToRead is the number of bytes to be read from the file
- LPDWORD lpNumberOfBytesRead is the number of bytes actually read
- LPOVERLAPPED lpOverlapped is used for concurrent processing.

The **WriteFile()** API is used to write data from buffer to file.

Syntax:

BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);

To write through the current size of file, the file must be opened with FILE_FLAG_WRITE_THROUGH option.

Lesson 15

Topic 15: Closing a File

After opening & using a file, it is required to close and invalidate the file handles in order to release the system resources.

The **CloseFile()** API is used to close a file. A file handle is passed as parameter to this API as a result of which the API will return True or False value. If the operation is successful, then it returns True value. In case if the handle is already invalid, then it will return False value.

Lesson 16

Lecture 16: Generic Characters

- Windows supports both ASCII (8-bit standard) and Unicode (16-bit standard) characters
- Windows have different API function variants to deal with both Unicode and ASCII characters.
- Unicode is used to represent characters in other languages like Arabic, Chinese, Urdu etc.
- The structure of a program is different for Unicode and ASCII characters.
- Generic program or code works for both the ASCII and Unicode.
- Windows provides a set of data types for ASCII, a set of data types for Unicode and a set of data types for Generic code like TCHAR, LPTSTR, LPCTST etc.
- For Unicode the following two macros should be used before including <windows.h>
    - #define UNICODE: it includes all the function headers
    - #define _UNICODE: it includes variable types
    - These macros will force the generic data type to be replaced by Unicode data types.

- sizeof() operator should be used in coding instead of hard coding if required.
- Use generic C Library functions like _itot(), _stprintf(), _tcscopy() etc.
- Use the _TEXT or T_macro for representing string constants.
- <tchar.h> file should be included for generic C functions.
- Most of the APIs in the latest version of Windows expect to pass the parameters in Unicode, therefore a generic code should be written instead of ASCII code.

Lesson 17

Lecture 17: Generic Functions

- Besides Generic data types, Windows also supports generic functions.
- Examples of generic functions are:

_tcscmp() instead of lstrcmp(), _tcscmpi() instead of lstrcmpi()

- Some functions that deal with Unicode characters and strings and work with locale settings transparently are:
    - CharUpper()
    - IsCharAlphaNumeric()
    - CompareString()
- Generic main() function is modified in terms of data types of its parameters and its name as mentioned below:

int _tmain(int argc, LPTSTR argv[])

- <windows.h> and <tchar.h> header files must be included before the main() function.
- Windows usually have two versions for each API function. For example, TextOut will have two variants i.e. TextOutA(16 bit API) and TextOutW(32 bit API).
- The use of Generic data types and Unicode macro enables the compiler to decide which version to choose.

Lesson 18

Lecture 18: Unicode Strategies

While writing a new code or enhancing an existing one, a programmer can adapt any of the following strategies based on requirements.

- Ignore Unicode Policy
- Use 8 bit only
- Ignore Unicode
- Use data types like char or CHAR
- Most of the Windows cannot be used ignoring unicode
- Use standard library functions like printf(), scanf(), atoi() etc.

Using Generic Code Policy

- The Unicode macro is used to switch between 8 bit and Unicode
- Generic functions are used
- Generic data types are used

Using Unicode Policy

- Use Unicode only
- Unicode functions are used
- Wide characters data types are used

Unicode and 8-bit Policy

- Handle Unicode and 8-bit
- Write code for both Unicode and 8-bit
- Decision regarding which option to choose is made at runtime
- Use of Unicode only strategy is now increasing popularly
- Use of generic code makes the code more flexible

Lesson 19

Lecture 19: Reporting Errors

- Windows Operating System provides the facility to report errors, if occurs.
- It is an important and salient feature of Windows to display error messages for user convenience.
- The important Windows function for reporting errors is GetLastError(). It returns the code for the last system error.
- Another function FormatMessage() translates the error code into meaningful English or language selected in preferences.
- A message string reference against the error code is stored in another parameter.
- Another function LocalFree() is also used with FormatMessage() to deallocate the allocated local memory.

Lesson 20

We will discuss an example of reporting errors. Windows have provide certain API's in which the functions of getlasterror() and formatmessage(), we will see how we can use these functions to report errors. For this we will make a generic function that we will use in reporting errors.

To write code we need following customized header files

- Enviornment.h
- Everything.h

Enviornment.h includes macros for the environment of the program which is basically used of UNICODE macro.

```
#if(WIN32_WINNT>=0x600)

#define WIN32_WINNT 0x600  /* Enable use of NT 5 (XP,2000,2003) functions */

#else

#if (WIN32_WINNT>=0x500 ) /*enable use of NT5(XP,2000,2003) functions */

#endif

#endif
```

In environment.h we have also

```
#ifdef UNICODE

#define_UNICODE

#endif

#ifndef UNICODE

#undef_UNICODE

#endif

#define LANG_DFLT LANG_ENGLISH  // set default language

#define SUBLANG_DFLT SUBLANG_ENGLISH_US
```

Everything.h includes all the header file that will be typically required for all the subsequent window programs.

```
#include "ENVIORNMENT.h

#include<windows.h>  // all headers

#include<tchar.h> // generic functions

#include<stdio.h>  // use for working in CLI

#include<stdlib.h>

#include<malloc.h>// memory allocation

#include<io.h>  // input output operations

#include<WinSock2.h> // for windows socket operations

#include "support.h"

#include _MT

#include <process.h> // for multi tasking and multi threading

#endif
```

Following is the code of reporting errors.

```
#include  "Everything.h"

VOID ReportError{LPCTSTR userMessage, DWORD exitCode, BOOL printErrorMessage}
```

```
{
DWORD eMsgLen, errNum=GetLastError();
LPSTR lpvSysMsg;
_ftprint(stderr,_t("%s\n"},userMessage};
If (printErrorMessage)
{
 eMsgLen= FormatMessage{FORMAT_MESSAGE_ALLOCATE_BUFFER|
FORMATE_MESSAGE_FROM_SYSTEM,
NULL,errNUM,MAKELANGID{LANG_NEUTRAL, SUBLANG_DEFAULT),
(LPSTR)&lpsSysMSg,0,NULL);
If(eMsgLen>0)
{
    _ftprint(stder,T("%s\n"|,lpvSysMsg));
}
Else
{
_ftprint{stderr,_T{"LastErrorNumber,%d\n"},errNum};
}
If (lpvSysMsg!=NULL) LOCALFree(lpvSysMsg);
}
If(exitCode>0)
ExitProcess(exitCode);
return;
}
```

Lesson 21

There are three standard IO devices used in Operating system.

- Input
- Output
- Error

An input operation is performed by default on standard IO devices. For example we does not mention in printf command where to print, it print the letter or any string by default on standard Output device and in getch, character is get by default input device. All the IO devices are manipulate in windows through handles.

Certain APIs are used to acquire handle to standard IO devices.

HANDLE GetStdHandle(DWORD nStdHandle);

- It returns a valid handle if the function succeed
- In case of failure it return INVALID_HANDEL_VALUE
- Successive calls to the functions will still run the same handle
- If the handle is closed it makes it subsequently unusable for the process in future.

Three types of values can be pass to the nSTDHandle

1. STD_INPUT_HANDLE
2. STD_OUPUT_HANDLE
3. STD_ERROR_HANDLE

STD_INPUT_HANDLE contains CONIN$(Console input) as an environment variable, STD_OUTPUT_HANDLE contains CONOUT$(Console Output) as an environment variable.

Operating system have also the concept of redirection using the given API

BOOL SetStdHandle(DWORD nStdHandle, HANDLE hHandle);

It also return true if calls succeed and return false in case of fail.

Lesson 22

We will see in this module how to display files using cosonle APIs of windows, in other words display file on screen is actually copying file on console. For this we made utility function "options()". We will also use this function further. This function take a variable list of parameters and use to parse these variable list. Basically we specify the list of parameters of a program on command prompt, there are number of options in it. Option () is use to parse these options. It identify the "-" prefix and check all the possible options, and set the flag against the set options.

Following is the code of the program

```
#include "Everything.h"

#include <stdarg.h>

DWORD Options (int argc, LPCTSTR argv [], LPCTSTR OptStr, …)   /*… show the
parameters list are variables */

{

   Va_list pFlagList;

LPBOOL pFlag;

Int iFlag = 0, iArg;

Va_start (pFlagList, OptStr);

While ( (pFlag= Va)arg (pFlagList, LPBOOL)) != NULL

          && iFlag<(int)_tcslen (OptStr)){

*pFlag= False;

 For (iArg= 1; !(*(pFlag) && iArg <argc &&argv[iARG] [0]== _T('-'); iArg++)

         *pFlag = _memtchr (argv [iArg], OptStr [iFlag], _tcslen (argv
[iArg]))!= NULL

iFlag++;

}
```

```
Va_end (pFlagList);

For (iArg= 1; !(*(pFlag) && iArg <argc &&argv[iARG] [0]== _T('-'); iArg++);

Retrun iArg;

}
```

This utility or program takes the number of parameters through command prompt, in which we specify option and file names. The Files names which are given , it open these file and try to print the content of these file on console and if no file name is given then it take the data from standard input and send to the standard output. It also use report error function in case of any error occur.

CatFile Function:

```
Static VOID CatFile (HANDLE hInFile, HANDLE hOutFile)

{

   DWORD nIn, nOut;

  BYTE buffer [BUF_SIZE];

While (ReadFile(hInFile, buffer, BUF_SIZE, $nIn, NULL) && (nIn !=0 )

&& WriteFile (hOutFile, buffer, nIn, &Out, NULL));

Return;

}
```

Lesson 23

Encryption is a very old technique, and Roman Empire use to encrypt secret conversation in war days and they use Caesar Cipher algorithm to encrypt. In this method an alphabet is substituted by another alphabet placed n positions forward in circular manner. The text that is changed using encryption method is called Cipher text.

The text that we are going to encrypt is called plain text so it is denoted by P and after encrypt we present it with C.

File Encryption Program

- if n=1 then A will be replaced by B, B will be replaced by C and so on uptil Z which will be again replaced by A.
- if n=2 then A will be replaced by C, B will be replaced by Dand so on uptil Y which will be again replaced by A
- it uses the function C= (P + n) mod 26

We take mod 26 because total character are 26.

We use mod 256 because of ASCII character

File Encryption Program

- The program uses a slightly different version of Ceasar Cipher adapted for ASCII characters
- C = (P + n) mod 256

This technique is not exactly cipher but little bit similar to cipher. Following is the code of encrypting file

```
#include "Everything.h"

#include <io.h>

BOOL cci_f (LPCTSTR, LPCTSTR, DWORD);

int _tmain (int argc, LPTSTR argv [])

{

   if (argc != 4)

         ReportError  (_T ("Usage: cci shift file1 file2"), 1, FALSE);



   if (!cci_f (argv [2], argv [3], _ttoi(argv[1])))

         ReportError (_T ("Encryption failed."), 4, TRUE);

   return 0;

}
```

```
BOOL cci_f (LPCTSTR fIn, LPCTSTR fOut, DWORD shift)

{

   HANDLE hIn, hOut;

   DWORD nIn, nOut, iCopy;

   BYTE buffer [BUF_SIZE], bShift = (BYTE)shift;

   BOOL writeOK = TRUE;

hIn = CreateFile (fIn, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hIn == INVALID_HANDLE_VALUE) return FALSE;

hOut = CreateFile (fOut, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hOut == INVALID_HANDLE_VALUE) {

         CloseHandle(hIn);

         return FALSE;

   }

   while (writeOK && ReadFile (hIn, buffer, BUF_SIZE, &nIn, NULL) && nIn > 0) {

         for (iCopy = 0; iCopy < nIn; iCopy++)

               buffer[iCopy] = buffer[iCopy] + bShift;

         writeOK = WriteFile (hOut, buffer, nIn, &nOut, NULL);

   }

   CloseHandle (hIn);

   CloseHandle (hOut);

   return writeOK;

}
```

Lesson 24

We will see in this module the types of API for file management. Windows provides lots of function for file and directory management. These functions are pretty straightforward and easy to use.

- Such functions perform operations like
  - Delete
  - Copy
  - Rename

Delete

Delete function will help to delete the file on a given path. For deleting file the following API is used.

BOOL DeleteFile(LPCTSTR lpFileName) ;

Returns TRUE if the file at the given valid file path is deleted

Copy

For copying file the following API is used .

BOOL CopyFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL bFailIfExists);

Copies an existing file from lpExistingFileName path to lpNewFileName, If new filename is same as existing then the file is overwritten only if bFailIfExists is false. It returns TRUE if the file at the given file is copied successfully

Hard Copy

Windows also provides hardlinks. Following API is used for creating hardlinks

BOOL CopyHardLink(LPCTSTR lpFileName, LPCTSTR lpExistingFileName, LPSECURITY_ATTRIBUTES lpSecurityAttributes);

- Creates a hard link for existing file
- Both the files must be on same system volume

- Security attributes will apply on new file name

Move

Use the following APIs to move files

BOOL MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName);

BOOL MoveFileEx(LPCWSTR lpExistingFileName, LPCWSTR lpNewFileName,

DWORD dwFlags);

MoveFile() fails if new file already exists. We use MoveFileEx() to overwrite existing file.

MoveFileEx() is the extended version of MoveFile() It can be used for both files and directories

Lesson 25

We will discuss the functions which we can use for directory management. We will do different directory operation like create directory , remove directory and move directory.

For **create directory** we will use the following function

BOOL CreateDirectory(LPCTSTR lpPathName,

LPSECURITY_ATTRIBUTES lpSecurityAttributes

);

For **remove directory** we will use the following function

BOOL RemoveDirectory(

LPCSTR lpPathName

);

- lpPathName specifies the path of the directory to be created or deleted.

Current Directory

In windows and many other operating system concept of current directory or current working directory is also exist. Each Process has a current working directory which can be changed/retrieved using the following API

```
BOOL SetCurrentDirectory(
  LPCTSTR lpPathName
);
DWORD GetCurrentDirectory(
  DWORD  nBufferLength,
  LPTSTR lpBuffer
);
```

Lesson 26

Console is the set of standard IO devices. Usually, a monitor and a keyboard are considered to be the console. The idea comes from virtual terminals used with mainframe computers

How to Perform IO on Console

We normally display something on console , or take some input from console or perform some operation on console Operating System treats input and out devices as file on console. Operating System regards console as a set of streams. The APIs used for File IO like ReadFile() and WriteFile() can be used to perform IO on Console. But well adapted APIs exist for console IO like ReadConsole() and WriteConsole.

Why used console based API

It is better to use console based APIs. They make use of generic strings (TCHAR) and not bytes.

They process characters based on current Console mode set by SetConsoleMode() API. This gives greater edge over processing

API's for Console IO

BOOL WINAPI SetConsoleMode(

HANDLE hConsoleHandle, // console handle

DWORD dwMode //specify mode

);

hConsoleHandle is the console handle which must have GENERIC_WRITE access

dwMode Can have any of the following values

ENABLE_ECHO_INPUT

Characters read by the ReadFile or ReadConsole function are written to the active screen buffer as they are read.

ENABLE_LINE_INPUT

The ReadFile or ReadConsole function returns only when a carriage return character is read.

ENABLE_PROCESSED_INPUT

If the ENABLE_LINE_INPUT mode is also enabled, backspace, carriage return, and line feed characters are handled by the system.

ENABLE_PROCESSED_OUTPUT

Backspace, tab, bell, carriage return, and line feed characters are processed.

ENABLE_WRAP_AT_EOL_OUTPUT

Line Wrapping is Enabled

Read Console API

BOOL WINAPI ReadConsole(

HANDLE hConsoleInput,

LPVOID lpBuffer,

DWORD nNumberOfCharsToRead,

LPDWORD lpNumberOfCharsRead,

LPVOID lpReserved

);

hConsoleInput [in]

A handle to the console input buffer. The handle must have the GENERIC_READ access right. For more information, see Console Buffer Security and Access Rights.

lpBuffer [out]

A pointer to a buffer that receives the data read from the console input buffer.

nNumberOfCharsToRead [in]

The number of characters to be read. The size of the buffer pointed to by the lpBuffer parameter should be at least nNumberOfCharsToRead * sizeof(TCHAR) bytes.

lpNumberOfCharsRead [out]

A pointer to a variable that receives the number of characters actually read.

lpReserved [in, optional]

This parameter can be NULL.

Lesson 27

In the previous module we see the certain APIs , which perform input/output operations on console, now we use these APIs. We create two types of utility functions, one is help us to display string on console and other is pass some message to user and also take input from users. Following are names and description of these functions.

Variable parameter list

●PrintStrings() function is designed which take variable list of parameters.

●Variable parameter list is processed using va_start(), va_arg()and va_end() library functions

Console Prompt functions:

- This function devised such that it prompt a given message to the user on console
- Further it also receives user response through the console

Print String Function:

```
#include "Everything.h"
#include <stdarg.h>
BOOL PrintStrings (HANDLE hOut, ...){
        DWORD msgLen, count;
        LPCTSTR pMsg;
        va_list pMsgList;       /* Current message string. */
        va_start (pMsgList, hOut);     /* Start processing msgs. */
        while ((pMsg = va_arg (pMsgList, LPCTSTR)) != NULL) {          // read variable list
        msgLen = lstrlen (pMsg);  // message length
if (!WriteConsole (hOut, pMsg, msgLen, &count, NULL) // in case of write console fail
                        && !WriteFile (hOut, pMsg, msgLen * sizeof (TCHAR),   &count,
NULL)) {
                va_end (pMsgList);
                return FALSE;
        }
        }
```

```
        va_end (pMsgList);

        return TRUE;

}

BOOL PrintMsg (HANDLE hOut, LPCTSTR pMsg){

        return PrintStrings (hOut, pMsg, NULL);

}
```

Console Prompt Function

```
BOOL ConsolePrompt (LPCTSTR pPromptMsg, LPTSTR pResponse, DWORD maxChar,
BOOL echo){

        HANDLE hIn, hOut;

        DWORD charIn, echoFlag;

        BOOL success;

// create console input file

  hIn = CreateFile (_T("CONIN$"), GENERIC_READ | GENERIC_WRITE, 0,

                NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// create console output file

        hOut = CreateFile (_T("CONOUT$"), GENERIC_WRITE, 0,

                NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

        /* Should the input be echoed? */

        echoFlag = echo ? ENABLE_ECHO_INPUT : 0;

   success =  SetConsoleMode (hIn, ENABLE_LINE_INPUT | echoFlag |
ENABLE_PROCESSED_INPUT)
```

```
&& SetConsoleMode (hOut, ENABLE_WRAP_AT_EOL_OUTPUT |
ENABLE_PROCESSED_OUTPUT)

        && PrintStrings (hOut, pPromptMsg, NULL)

        && ReadConsole (hIn, pResponse, maxChar - 2, &charIn, NULL);


    /* Replace the CR-LF by the null character. */

    if (success)

    pResponse [charIn - 2] = _T('\0');

    else

    ReportError (_T("ConsolePrompt failure."), 0, TRUE);

    CloseHandle (hIn);

    CloseHandle (hOut);

    return success;

}
```

Lesson 28

In this module we will see a small code which will use get current directory.

Printing Current Directory

This example uses the ***GetCurrentDirectory()*** function to get the current directory, further it uses the console IO to print the path.

```
#include "Everything.h"
```

```
#define DIRNAME_LEN (MAX_PATH + 2)

int _tmain (int argc, LPTSTR argv [])

{



    TCHAR pwdBuffer [DIRNAME_LEN];

    DWORD lenCurDir;

//calling getcurrent directory and name stored in pwdbuffer

 lenCurDir = GetCurrentDirectory (DIRNAME_LEN, pwdBuffer);

    if (lenCurDir == 0)  // in case of failure

            ReportError (_T ("Failure getting pathname."), 1, TRUE);

    if (lenCurDir > DIRNAME_LEN)

            ReportError (_T ("Pathname is too long."), 2, FALSE);

       PrintMsg (GetStdHandle (STD_OUTPUT_HANDLE), pwdBuffer);  // in case of correction

    return 0;

}
```

Lesson 29

If we see historically, there were some file system which was of 12-bit after that we have 32-bit system and still somewhere 32-bit file system are used. FAT based system allowed a maximum file size of 2^{32} bytes which is 4GB. NTFS theoretically provides the file size limit of 2^{64} which is very huge.

Files of such proportion are called huge files. Although for most of the application 32 bit file space is sufficient. However due to rapid technological changes leading to increased disk spaces its

useful to know how to deal with **64 bit** huge file spaces and windows facilitate with some API's that support 64-bit file system.

Lesson 30

File Pointer APIs

Whenever a file is opened using CreateFile() the file pointer is placed at the start of file. The file pointer changes as ReadFile() or WriteFile() operations are performed. Every subsequent read/write operation is performed at the current file pointer position.

SetFilePointer() and **SetFilePointerEx()** functions are used to change file pointer position. For random file accessSetFilePointer() is clumsy to used for 64 bit operations. SetFilePointerEx() can be more readily used for 64 bit operations

Use of SetFilePointer() used is difficult as compared to SetFilePointerEx().

SetFilePointer()

```
DWORD SetFilePointer(
  HANDLE hFile,
  LONG   lDistanceToMove,  // for 32-bit file
  PLONG  lpDistanceToMoveHigh,  // pointer to a long for NTFS
  DWORD  dwMoveMethod
);
```

hFile

A handle to the file.

lDistanceToMove

The low order 32-bits of a signed value that specifies the number of bytes to move the file pointer.

lpDistanceToMoveHigh

A pointer to the high order 32-bits of the signed 64-bit distance to move.

dwMoveMethod

The starting point for the file pointer move.

FILE_BEGIN // file pointer move number of bytes w.r.t the start of file

FILE_CURRENT // file pointer move with respect to the current position

FILE_END // file pointer move w.r.t the end of file

Lesson 31

For large files that may have size 2^{64} , we need to understand the 64 bit arithmetic. To facilitate 64-bit integer arithmetic windows provide a union LARGE_INTEGER. This union has structure for dealing with **lower and higher double** words Moreover it also has a field to deal with whole quadword of type LONGLONG.

Definition of Structure of large integer

```
typedef union _LARGE_INTEGER {
  struct {
        DWORD LowPart;
        LONG  HighPart;
  } ;
  struct {
        DWORD LowPart;
        LONG  HighPart;
  } u;
  LONGLONG QuadPart;
```

} LARGE_INTEGER;

Extension of SetFilePointerEx

BOOL SetFilePointerEx(

HANDLE hFile,

LARGE_INTEGER liDistanceToMove,

PLARGE_INTEGER lpNewFilePointer,

DWORD dwMoveMethod

);

hFile

A handle to the file. The file handle must have been created with the GENERIC_READ or GENERIC_WRITE access right

liDistanceToMove

The number of bytes to move the file pointer.

lpNewFilePointer

A pointer to a variable to receive the new file pointer.

dwMoveMethod

he starting point for the file pointer move.

FILE_BEGIN

FILE_CURRENT

FILE_END

Lesson 32

Overlap structure is a structure which is defined in the 'windows.h' and can be used with read file and write file API's. This structure can also be used to set file pointer position, Although the name is overlapped but it is not necessary that can used only for overlapped operations, we also used in multitasking.

SetFilePointer() or SetFilePointerEx() need not be invoked we can achieve this goal using readfile and write file. Also, you can append to the file by specifying 0xffffffff in both Offset and OffsetHigh fields

Definition of Overlap structure:

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;  // reserved filed
  ULONG_PTR InternalHigh;    // reserved filed
  union {
        struct {
        DWORD Offset;
        DWORD OffsetHigh;
        } DUMMYSTRUCTNAME;
        PVOID Pointer;
  } DUMMYUNIONNAME;
  HANDLE     hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Implementation:

```
filePos.QuadPart=x;   // large integer variable
ov.Offset=filePos.LowPart;   // place low part of file position
```

ov.OffsetHigh=filePos.HighPart; // place high part of file position

ReadFile(hFile, buf, sizeof(buf), &nRead, &ov);

…..

…

WriteFile(hFile, buf, sizeof(buf), &nWrite, &ov);

Lesson 33

One method of getting file size is already exist and that is to open a file first using create file, once file is open the file pointer is pointing to the first byte then we move file pointer to the end of file (eof). So file pointer is move from starting to end of file is give use the size of the file. Windows also provides an API to get file size GetFileSizeEx()

GetFileSizeEx()

BOOL GetFileSizeEx(

  HANDLE   hFile,

  PLARGE_INTEGER lpFileSize

);

hFile

A handle to the file. The handle must have been created with the FILE_READ_ATTRIBUTES access right

lpFileSize

A pointer to a LARGE_INTEGER structure that receives the file size, in bytes.

Windows also give option to set the size. The file size can also be changed, reducing the file size truncate data. Increasing the file size can be useful where the size of file is expected to grow. We use SetEndOfFileEx() to change the file size.

Lesson 34

In this topic we discuss the example of creates a file with a capacity of specified records. The file has a header followed by equal size records. The feature of this example is, user can modify any record randomly and get the total count of records in the file.

```
#include "Everything.h"

        #define STRING_SIZE 256

        typedef struct _RECORD { /* File record structure */

                DWORD                   referenceCount;  /* 0 meands an empty record */

                SYSTEMTIME              recordCreationTime;

                SYSTEMTIME              recordLastRefernceTime;

                SYSTEMTIME              recordUpdateTime;

                TCHAR                   dataString[STRING_SIZE];

        } RECORD;

        typedef struct _HEADER { /* File header descriptor */

                DWORD                   numRecords;

                DWORD                   numNonEmptyRecords;

        } HEADER;

        int _tmain (int argc, LPTSTR argv[])

        {
```

```
HANDLE hFile;

LARGE_INTEGER currentPtr;

DWORD OpenOption, nXfer, recNo;

RECORD record;

TCHAR string[STRING_SIZE], command, extra;

OVERLAPPED ov = {0, 0, 0, 0, NULL}, ovZero = {0, 0, 0, 0, NULL};

HEADER header = {0, 0};

SYSTEMTIME currentTime;

BOOLEAN headerChange, recordChange;


int prompt = (argc <= 3) ? 1 : 0;

 if (argc < 2)

        ReportError (_T("Usage: RecordAccess file [nrec [prompt]]"), 1, FALSE);

OpenOption = ((argc > 2 && _ttoi(argv[2]) <= 0) || argc <= 2) ? OPEN_EXISTING : CREATE_ALWAYS;

        hFile = CreateFile (argv[1], GENERIC_READ | GENERIC_WRITE,

                        0, NULL, OpenOption, FILE_FLAG_RANDOM_ACCESS, NULL);

        if (hFile == INVALID_HANDLE_VALUE)

        ReportError (_T("RecordAccess error: Cannot open existing file."), 2, TRUE);

        if (argc >= 3 && _ttoi(argv[2]) > 0)  {

                        header.numRecords = _ttoi(argv[2]);

                        if (!WriteFile(hFile, &header, sizeof (header), &nXfer, &ovZero))

                                ReportError (_T("RecordAccess Error: WriteFile header."), 4, TRUE);
```

```
currentPtr.QuadPart = (LONGLONG)sizeof(RECORD) * _ttoi(argv[2]) + sizeof(HEADER);

if (!SetFilePointerEx (hFile, currentPtr, NULL, FILE_BEGIN))

ReportError (_T("RecordAccess Error: Set Pointer."), 4, TRUE);

if (!SetEndOfFile(hFile))

ReportError (_T("RecordAccess Error: Set End of File."), 5, TRUE);

if (prompt) _tprintf (_T("Empty file with %d records created.\n"), header.numRecords);

return 0;

}

if (!ReadFile(hFile, &header, sizeof (HEADER), &nXfer, &ovZero))

ReportError (_T("RecordAccess Error: ReadFile header."), 6, TRUE);

if (prompt) _tprintf (_T("File %s contains %d non-empty records of size %d.\n Total capacity: %d\n"),

argv[1], header.numNonEmptyRecords, sizeof(RECORD), header.numRecords);

while (TRUE) {

headerChange = FALSE; recordChange = FALSE;

if (prompt) _tprintf (_T("Enter r(ead)/w(rite)/d(elete)/qu(it) record#\n"));

_tscanf (_T("%c%u%c"), &command, &recNo, &extra);

if (command == _T('q')) break;

if (recNo >= header.numRecords) {

if (prompt) _tprintf (_T("record Number is too large. Try again.\n"));

continue;
```

```
                    }
                     currentPtr.QuadPart = (LONGLONG)recNo * sizeof(RECORD) +
sizeof(HEADER);
                    ov.Offset = currentPtr.LowPart;
                    ov.OffsetHigh = currentPtr.HighPart;
                    if (!ReadFile (hFile, &record, sizeof (RECORD), &nXfer, &ov))
                            ReportError (_T("RecordAccess: ReadFile failure."), 7, FALSE);
                    GetSystemTime (&currentTime); /* Use to update record time fields */
                    record.recordLastRefernceTime = currentTime;
if (command == _T('r') || command == _T('d')) { /* Report record contents, if any */
                            if (record.referenceCount == 0) {
                                    if (prompt) _tprintf (_T("record Number %d is empty.\n"),
recNo);
                                    continue;
                            } else {
                                     if (prompt) _tprintf (_T("record Number %d. Reference
Count: %d \n"),
                                             recNo, record.referenceCount);
                                    if (prompt) _tprintf (_T("Data: %s\n"), record.dataString);
                                    /* Exercise: Display times. See ls.c for an example */
                            }
                            if (command == _T('d')) { /* Delete the record */
                                    record.referenceCount = 0;
                                    header.numNonEmptyRecords--;
```

```
                headerChange = TRUE;
                recordChange = TRUE;
            }
} else if (command == _T('w')) { /* Write the record, even if for the first time */
            if (prompt) _tprintf (_T("Enter new data string for the record.\n"));
            _fgetts (string, sizeof(string), stdin); // Don't use _getts (potential buffer overflow)
            string[_tcslen(string)-1] = _T('\0'); // remove the newline character
            if (record.referenceCount == 0) {
                record.recordCreationTime = currentTime;
                header.numNonEmptyRecords++;
                headerChange = TRUE;
            }
            record.recordUpdateTime = currentTime;
            record.referenceCount++;
            _tcsncpy (record.dataString, string, STRING_SIZE-1);
            recordChange = TRUE;
} else {
             if (prompt) _tprintf (_T("command must be r, w, or d. Try again.\n"));}
          /* Update the record in place if any record contents have changed. */
           if (recordChange && !WriteFile (hFile, &record, sizeof (RECORD), &nXfer, &ov))
             ReportError (_T("RecordAccess: WriteFile update failure."), 8, FALSE);
```

```
                    /* Update the number of non-empty records if required */

                    if (headerChange) {

                            if (!WriteFile (hFile, &header, sizeof (header), &nXfer, &ovZero))

                                    ReportError (_T("RecordAccess: WriteFile update failure."),
9, FALSE);

                    }

            }

             if (prompt) _tprintf (_T("Computed number of non-empty records is: %d\n"),
header.numNonEmptyRecords);

            if (!ReadFile(hFile, &header, sizeof (HEADER), &nXfer, &ovZero))

                    ReportError (_T("RecordAccess Error: ReadFile header."), 10, TRUE);

             if (prompt) _tprintf (_T("File %s NOW contains %d non-empty records.\nTotal
capacity is: %d\n"),

                    argv[1], header.numNonEmptyRecords, header.numRecords);

            CloseHandle (hFile);

            return 0;

    }
```

Lesson 35

Windows provide a certain set of APIs for search files/folders within the hierarchical structure of Directories/folders. These APIs include:

FindFirstFile(), FindFirstFileEx(), FindNextFile() and FindClose(). Detail of these APIs are:

FindFirstFile() API

Handle FindFirstFile(LPCSTR lpFileName, LPWIN32_FIND_DATA ***A lpFindFileData);***

Where **lpFileName** represents the directory or path, and the filename. The name can include wildcard characters, for example, an asteristk (*) or a question mark (?).

lpFindFileData: A pointer to the WIN32_FIND_DATA structure that receives information about a found file or directory.

Structure or Detail of WIN32_FIND_DATA structure:

```
typedef struct  _WIN32_FIND_DATA    {
     DWORD  dwFileAttributes;
    FILETIME  ftCreationTime;
    FILETIME  ftLastAccessTime;
    FILETIME  ftLastWriteTime;
     DWORD   nFileSizeHigh;
     DWORD  nFileSizeLow;
     DWORD  dwReserved0;
     DWORD  dwReserved1;
     CHAR cFileName[Max_Path];
     CHAR cAlternateFileName[14];
     DWORD  dwFileType;
     DWORD  dwCreatorType;
     DWORD   WFinderFlags;
};
```

FindNextFile() API:

BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA *A lpFindFileData*);

Where hFindFile represents the search handle returned by a previous call to the FindFirstFile or FindFirstFileEx function &

lpFindFileData is a pointer to the WIN32_FIND_DATA structure that receives information about the found file or sub-directory.

FindClose() API

BOOL FindClose(HANDLE hFindFile);

Lesson 36

Certain other APIs are also used for getting the file attributes but these API need to have an open file handle rather than scan a directory or use a filename.

GetFileTime() API

BOOL GetFileTime(HANDLE hFile, LPFILETIME lpCreationTime, LPFILETIME lpLastAccessTime, LPFILETIME lpLastWriteTime);

Where **hFile** is a handle to the file or directory.

lpCreationTime is a pointer to a FILETIME structure to receive the data and time the file or directory was created.

lpLastAccessTime is a pointer to a FILETIME structure to receive the data and time the file or directory was last accessed.

lpLastWriteTime is a pointer to a FILETIME structure to receive the data and time the file or directory was last written to truncated or overwritten.

FileTimeToSystemTime() API

It converts the file time into system time.

SystemTimeToFileTime() API

It converts the system time into file time.

CompareFileTime() API

It compares file times of two files. It returns -1 if less, 0 if equal and +1 if greater.

SetFileTime() API

It sets the three time of file. NULL used if the file time is not to be changed.

FileTimeToLocalFileTime() and LocalFileTimeToFileTime() APIs converts UTC and local File time.

GetFileType API

It provides information regarding the type of file (disk file or pipes)

GetFileAttributes() API

DWORD GetFileAttributes(LPCTSTR lpFileName);

Where lpFileName is the name of a file or directory. Its return value is:

FILE_ATTRIBUTE_DIRECTORY, FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_NORMAL, FILE_ATTRIBUTE_TEMPORARY.

In case of failure, the return value is INVALID_FILE_ATTRIBUTES whose value is 0xFFFFFFFF.

Lesson 37

Topic 37: Temporary File Names

Windows provide the facility of creating temporary files for storing the intermediate results. These files are assigned unique names in a directory with extension .tmp. Certain APIs are used for creating temporary files. These include:

GetTempFileName API

UINT GetTempFileName(LPCTSTR lpPathName, LPCTSTR lpPrefixString, UINT uUnique, LPSTR lpTempFileName);

Where **lpPathName** represents the directory path for the filename. The string cannot be longer than 14 characters.

lpPrefixString represents the null-terminated prefix string. The first 3 characters of this string is used as a prefix of the filename.

uUnique represents an unsigned integer to be used in creating the temporay filename.

lpTempFileName is a pointer to the buffer that receives the temporary filename

Lesson 38

Topic 38: Listing File Attributes

We can get the attributes of a file, listing of files and can traverse the directory structure using certain windows APIs.

An application called **lsW** is used for showing files and listing their attributes. It uses two option switched that is –l and –R where –l option is used to list the attributes of files in a folder and –R is used for recursive traversal through subfolders.

This application or program will work with a relative pathname; it will not work with absolute pathname.

Program: File Listing and Directory Traversal

```
#include<everything.h>

BOOL TraverseDirectory(LPTSTR, LPTSTR, DWORD, LPBOOL);

DWORD FileType(LPWIN32_FIND_DATA);

BOOL ProcessItem(LPWIN32_FIND_DATA, DWORD, LPBOOL);

Int  _tmain(int argc, LPTSTR argv[])

{
```

```
 BOOL flags[MAX_OPTIONS], ok=TRUE;

 TCHAR searchPattern[MAX_PATH +1], currPath[MAX_PATH_LONG +1], parrentPath[MAX_PATH_LONG
+1];

LPTSTR pSlash, pSearchPattern;

int i, fileIndex;

DWORD pathLength;

fileIndex = Options(argc, argv, _T("RI"), &flags[0], &flags[1], NULL);

/* parse the search pattern into two parts: the parent and the filename or wild card expression. The
filename is the longest suffix not containing a slash. The parent is the remaining prefix with a slash. This
is performed for all command line search pattern. If no file is specified, use * as the search pattern */

pathLength = getCurrentDirectory(MAX_PATH_LONG, currPath);

if(pathLength==0 || pathLength>= MAX_PATH_LONG)

{   /* pathLength>= MAX_PATH_LONG (32780) should be impossible  */

 ReportError(_T("GetCurrentDirectory failed", 1, TRUE);
```

Lesson 39

UNIX Touch command changes file access and changes the time to current system time.

For changing file time the GetSystemTimeAsFileTime is more convenient than calling Get SystemTime followed by SystemTimeToFileTime.

These two functions are provided by sysinfoapi.h api used to retrieve system time.

GetSystemTimeAsFileTime() take file pointer as argument and retrieves the current system date and time in Coordinated Universal Time (UTC) format.

GetSystemTime() A pointer to a SYSTEMTIME structure to receive the current system date and time. Retrieves the current system date and time in Coordinated Universal Time (UTC) format.

SetFileTime() Sets the date and time that the specified file or directory was created, last accessed, or last modified.

Lesson 40

C Library for file processing have following features:

- Code is portable to Operating system running other than windows.
- Convenient line and character oriented functions are simply string processing.
- The functions are higher level and easy to use than windows functions.
- Line and stream character-oriented functions can be migrated to generic calls.

C library also have some limitations but some performance advantages are also present when compared to windows functions.

Lesson 41

Windows OS can lock files completely or some part of a file.There are two functions avalible LockfileEx and LockFile.

The syntax and parameters list of LockfileEx is given below:

BOOL LockFileEx(

HANDLE hFile,

DWORD dwFlags,

DWORD dwReserved,

DWORD nNumberOfBytesToLockLow,

DWORD nNumberOfBytesToLockHigh,

LPPVERLAPPED lpOverlapped

)

To Unlock a file UnlockFileEx() function is used.Parameters list is highly similar to LockFileEx.

Lesson 42

If a program does not release a lock or holds the lock longer, other programs will

not be able to proceed and their performance will be negatively impacted.

Therefore, programs should be designed in such a way that locks are released after the usage is completed using appropriate functions.

Lesson 43

- File locking can generate deadlocks just like deadlocks generated by mutual exclusion locks.
- A read or write may be able to complete its request before encountering a conflicting lock.
- The read or write will return false, and the byte transfer count will be less than the number requested.
- Shared locks if not used properly also generate unexpected errors, as if one process have a shared lock on a file other processes can only ready the file but cannot write into it.

Lesson 44

The Windows Registry is a hierarchical database that stores low level settings for the Microsoft Windows operating system and for applications that opt to use the registry. The OS kernel, device drivers, services, Security Accounts Manager, and user interfaces all can read and update the registry values.

In other words, the registry or Windows Registry contains different information e.g. settings, options, and other values for programs and hardware installed on the system. For example, when a program is installed, a new key containing settings such as its version is added to the Windows Registry.

Lesson 45

The registry contains two basic elements: keys and values. Registry keys are container objects similar to folders. Registry values are non-container objects similar to files. Keys may contain values and subkeys. Keys are referenced with a syntax similar to Windows' path names, using backslashes to indicate levels of hierarchy. Keys must have a case insensitive name without backslashes.

The hierarchy of registry keys can only be accessed from a known root key handle (which is anonymous but whose effective value is a constant numeric handle) that is mapped to the content of a registry key preloaded by the kernel from a stored "hive", or to the content of a subkey within another root key, or mapped to a registered service or DLL that provides access to its contained subkeys and values.

E.g. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows refers to the subkey "Windows" of the subkey "Microsoft" of the subkey "Software" of the HKEY_LOCAL_MACHINE root key.

There are seven predefined root keys, traditionally named according to their constant handles defined in the Win32 API, or by synonymous abbreviations (depending on applications):[4]

- HKEY_LOCAL_MACHINE or HKLM
- HKEY_CURRENT_CONFIG or HKCC
- HKEY_CLASSES_ROOT or HKCR
- HKEY_CURRENT_USER or HKCU
- HKEY_USERS or HKU
- HKEY_PERFORMANCE_DATA (only in Windows NT, but invisible in the Windows Registry Editor)[5]

HKEY_DYN_DATA (only in Windows 9x, and visible in the Windows Registry Editor)

Lesson 46

Service Manager stores many settings in the registry. You seldom have to edit the registry yourself, because most of those settings are derived from entries that you make in day-to-day use. However, some changes to settings might occasionally be required. Service Manager stores most registry values in the following locations:

- HKEY_CURRENT_USER\Software\Microsoft\System Center<version>\Service Manager\Console
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\System Center<version>

Lesson 47

You can show all items directly within a registry key by using Get-ChildItem. Add the optional Force parameter to display hidden or system items. For example, this command displays the items directly within PowerShell drive HKCU:, which corresponds to the HKEY_CURRENT_USER registry hive:

PowerShell

```
Get-ChildItem -Path HKCU:\ | Select-Object Name
```

You can also specify this registry path by specifying the registry provider's name, followed by ::. The registry provider's full name is Microsoft.PowerShell.Core\Registry, but this can be shortened to just Registry. Any of the following commands will list the contents directly under HKCU:.

PowerShell

```
Get-ChildItem -Path Registry::HKEY_CURRENT_USERGet-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USERGet-ChildItem -Path Registry::HKCUGet-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKCUGet-ChildItem HKCU:
```

Lesson 48

Structured exception handling (SEH) is a Microsoft extension to C to handle certain exceptional code situations, such as hardware faults, gracefully. Although Windows and Microsoft C++ support SEH, we recommend that you use ISO-standard C++ exception handling. It makes your code more portable and flexible. However, to maintain existing code or for particular kinds of programs, you still might have to use SEH.

Lesson 49

Different languages have different implementation for exception handling.
For example Python handles the exception following way:

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The else block lets you execute code when there is no error.

The finally block lets you execute code, regardless of the result of the try- and except blocks.

Lesson 50

Writing an exception filter in windows

You can handle an exception either by jumping to the level of the exception handler or by continuing execution. Instead of using the exception handler code to handle the exception and falling through, you can use a *filter* expression to clean up the problem. Then, by returning *EXCEPTION_CONTINUE_EXECUTION* (-1), you may resume normal flow without clearing the stack.

It's a good idea to use a function call in the *filter* expression whenever *filter* needs to do anything complex. Evaluating the expression causes execution of the function, in this case, *Eval_Exception*.

This handler passes control to another handler unless the exception is an integer or floating-point overflow. If it is, the handler calls a function (*ResetVars* is only an example, not an API function) to reset some global variables. The *__except* statement block, which in this example is empty, can never be executed because *Eval_Exception* never returns *EXCEPTION_EXECUTE_HANDLER* (1).

Using a function call is a good general-purpose technique for dealing with complex filter expressions. Two other C language features that are useful are:

The conditional operator

The comma operator

Lesson 51

Language like python have different type of exception codes some are given as follows:

OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.

SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.

Lesson 52

Exception Handling sequence for python is given below:

try:

You do your operations here;

.....

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

.....

else:

If there is no exception then execute this block.

Lesson 53

IEEE defines a standard for floating-point exceptions it is called IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754). This standard have defined five types of floating-point exception:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact calculation

Lesson 54

The key differences between exceptions and errors are as follows:

Errors	Exception
Errors are usually raised by the environment in which the application is running.	Exceptions are caused by the code of the app the code belongs to.
It is not possible to recover from an error.	The use of try-catch blocks can handle exceptions and recover the system from exception.
Errors occur at run-time and are unknown by the compiler.	Exceptions may or may not be caught by the compiler.
Programmers include an explicit test to check for error,for example whether a file read/write operation has failed.	An exception could occur nearly anywhere, and it is practical to test for an exception.

Lesson 55

Programmer can use ReportError function to report the error and let windows treat it and terminate the process.But it have its own limitations.

• A fatal error terminates the entire process when only a single thread should terminate.

• The programmer may require to continue program execution rather than terminate the process.

• Synchronization resources, such as events or semaphores, will not be released in most of the circumstances.

Lesson 56

Termination handler provides same functionality as an exception handler,but it is executed when a thread leaves a block as a result normal program flow as well as when an exception occurs. A termination handler cannot diagnose an exception.

A termination handler consists of the following elements.

- A guarded body of code
- A block of termination code to be executed when the flow of control leaves the guarded body

Termination handlers are declared in language-specific syntax. Using the Microsoft C/C++ Optimizing Compiler, they are implemented using __try and __finally.

The guarded body of code can be a block of code, a set of nested blocks, or an entire procedure or function.

The termination block is executed when the flow of control leaves the guarded body, regardless of whether the guarded body terminated normally or abnormally.

Lesson 57

Termination and exception handlers allow you to make your program more robust by both simplifying recovery from errors and exceptions and helping to ensure that resources and file locks are freed at critical junctures.

Consider the below given code where we are checking for invalid handle and ReportingException.

If (hin == INVALID_HANDLE_VALUE)

ReportException(argv[iFile],1);

If(!GetFileSizeEx(hIn, &fsize) || fsize.HighPart > 0)

ReportException(_T("FIle is Too Large",1));

Lesson 58

The filter function identifies the exception type and based on the type of the exception, the handler can treat each exception differently. In the following program, the exception handling and termination of a program are illustrated using a filter function.

```
#include "Everything.h"
#include <float.h>

DWORD Filter(LPEXCEPTION_POINTERS, LPDWORD);
double x = 1.0, y = 0.0;

int _tmain(int argc, LPTSTR argv[])
{
    DWORD eCategory, i = 0, ix, iy = 0;
    LPDWORD pNull = NULL;
    BOOL done = FALSE;
    DWORD fpOld, fpNew;
    fpOld = _controlfp(0, 0); /* Save old control mask. */
                    /* Enable floating-point exceptions. */
    fpNew = fpOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT
          | EM_ZERODIVIDE | EM_DENORMAL | EM_INVALID);
    _controlfp(fpNew, MCW_EM);

    while (!done) __try { /* Try-finally. */
        _tprintf(_T("Enter exception type: "));
        _tprintf(_T
              (" 1: Mem, 2: Int, 3: Flt 4: User 5: __leave "));
        _tscanf(_T("%d"), &i);
        __try { /* Try-except block. */
            switch (i) {
            case 1: /* Memory reference. */
                ix = *pNull; *pNull = 5; break;
            case 2: /* Integer arithmetic. */
                ix = ix / iy; break;
            case 3: /* Floating-point exception. */
                x = x / y;
                _tprintf(_T("x = %20e\n"), x); break;
            case 4: /* User-generated exception. */
                ReportException(_T("User exception"), 1); break;
            case 5: /* Use the _leave statement to terminate. */
```

```
            __leave;
         default: done = TRUE;
         }
      } /* End of inner __try. */

      __except (Filter(GetExceptionInformation(), &eCategory))
      {
         switch (eCategory) {
            case 0:
               _tprintf(_T("Unknown Exception\n")); break;
            case 1:
               _tprintf(_T("Memory Ref Exception\n")); continue;
            case 2:
               _tprintf(_T("Integer Exception\n")); break;
            case 3:
               _tprintf(_T("Floating-Point Exception\n"));
               _clearfp(); break;
            case 10:
               _tprintf(_T("User Exception\n")); break;
            default:
               _tprintf( _T("Unknown Exception\n")); break;
         } /* End of switch statement. */

         _tprintf(_T("End of handler\n"));
      } /* End of try-except block. */
   } /* End of While loop -- the termination handler is below. */

   __finally { /* This is part of the while loop. */
      _tprintf(_T("Abnormal Termination?: %d\n"),
            AbnormalTermination());
   }
   _controlfp(fpOld, 0xFFFFFFFF); /* Restore old FP mask.*/
   return 0;
}
```

The program generates an exception based on the type of the exception entered by the user. The floating point exceptions are enabled with the help of controlfp function and old status is saved in the fpOld. The try block mentions the different cases of exception generation with the help of a switch statement.

In the except clause, there is a filter function calling another function GetExceptionInformation(). eCategory is a reference variable whose value determines the type of

exception being thrown. For the outer try, there is a finally clause used for any abnormal termination of the program. The old value of the floating point mask is also restored at this stage.

Now we will see how the values in the ecategory reference variable are placed.

```
static DWORD Filter(LPEXCEPTION_POINTERS pExP, LPDWORD eCategory)
/* Categorize the exception and decide action. */
{
    DWORD exCode, readWrite, virtAddr;
    exCode = pExP->ExceptionRecord->ExceptionCode;
    _tprintf(_T("Filter. exCode: %x\n"), exCode);
    if ((0x20000000 & exCode) != 0) { /* User exception. */
        *eCategory = 10;
        return EXCEPTION_EXECUTE_HANDLER;
    }

    switch (exCode) {
        case EXCEPTION_ACCESS_VIOLATION:
            readWrite = /* Was it a read, write, execute? */
                  pExP->ExceptionRecord->ExceptionInformation[0];
            virtAddr = /* Virtual address of the violation. */
                  pExP->ExceptionRecord->ExceptionInformation[1];
            _tprintf(
            _T("Access Violation. Read/Write/Exec: %d. Address: %x\n"),
                  readWrite, virtAddr);
            *eCategory = 1;
            return EXCEPTION_EXECUTE_HANDLER;
        case EXCEPTION_INT_DIVIDE_BY_ZERO:
        case EXCEPTION_INT_OVERFLOW:
            *eCategory = 2;
            return EXCEPTION_EXECUTE_HANDLER;
        case EXCEPTION_FLT_DIVIDE_BY_ZERO:
        case EXCEPTION_FLT_OVERFLOW:
            _tprintf(_T("Flt Exception - large result.\n"));
            *eCategory = 3;
            _clearfp();
            return EXCEPTION_EXECUTE_HANDLER;

        default:
            *eCategory = 0;
            return EXCEPTION_CONTINUE_SEARCH;
    }
}
```

User generated exceptions are identified based on the masking operation generating zero as a result.

Lesson 59

Console control handlers are quite similar to the mechanism of exception handlers. Normal exceptions respond to several asynchronous events like division by zero, invalid page fault etc., but they do not respond to console related events like Ctrl+C. Console control handlers can detect and handle the events that are console related. The API SetConsoleCtrlHandler() is used to add console handlers.

BOOL SetConsoleCtrlHandler(PHANDLER_ROUTINE HandlerRoutine, BOOL Add);

The API takes the address of the HandlerRoutine and Add Boolean as parameters. There can be a number of handler routines if the Add parameter is set as TRUE. If the HandlerRoutine parameter is set as NULL and Add is TRUE, then the Ctrl-C signal will be ignored.

BOOL HandlerRoutine (DWORD dwCtrlType);

The return type of HanlderRoutine should be Boolean taking only one parameter of type DWORD. The HandlerRoutine() function takes a DWORD as a parameter and returns a Boolean value.

dwCtrlType identifies the signal and its values can be:

1. CTRL_C_EVENT: Indicates that a Ctrl-C sequence was entered through keyboards.
2. CTRL_CLOSE_EVENT: Indicates that the console window is being closed.
3. CTRL_BREAK_EVENT: Indicates Ctrl-Break sequence.
4. CTRL_LOGOFF_EVENT: Indicates the user is trying to logoff.
5. CTRL_SHUTDOWN_EVENT: Indicates the user is trying to shut down.

The handler routine will be invoked if a console exception is detected. Handler routine runs in an independent thread within the process. Raising an exception within the handler routine will not interfere with the working of the original routine that created the handler routine. Signals apply to the whole process, while exception applies to a single thread.

Usually signal handlers are used to perform cleanup tasks whenever a shutdown, close or logoff events are detected. Signal Handler would return a TRUE value in case it takes care of the task or it may return FALSE. In case of FALSE, the next handler in the chain is invoked. Signal

handler chain is invoked in the reverse order of which they are set up in and the system signal handler is the last one in this chain.

Lesson 60

We have a simple program that set up a console control handler and starts beeping in a loop. The control handler is invoked whenever a console event occurs. The handler handles the event likewise and clears an exitFlag to end the loop of the main function.

```
/* Chapter 4. CNTRLC.C */

/* Catch Cntrl-C signals. */

#include "Everything.h"

static BOOL WINAPI Handler(DWORD cntrlEvent);

static BOOL exitFlag = FALSE;

int _tmain(int argc, LPTSTR argv[])



/* Beep periodically until signaled to stop. */

{

                /* Add an event handler. */

        if (!SetConsoleCtrlHandler(Handler, TRUE))

                ReportError(_T("Error setting event handler"), 1, TRUE);



        while (!exitFlag) { /* This flag is detected right after a beep, before a handler exits */

                Sleep(4750); /* Beep every 5 seconds; allowing 250 ms of beep time. */
```

```
                Beep(1000 /* Frequency */, 250 /* Duration */);
        }
}
        _tprintf(_T("Stopping the main program as requested.\n"));
        return 0;
}
BOOL WINAPI Handler(DWORD cntrlEvent)
{
        switch (cntrlEvent) {
                /* The signal timing will determine if you see the second handler message
*/
                case CTRL_C_EVENT:
                        _tprintf(_T("Ctrl-C received by handler. Leaving in 5 seconds or
less.\n"));
                        exitFlag = TRUE;
                        Sleep(4000); /* Decrease this time to get a different effect */
                        _tprintf(_T("Leaving handler in 1 second or less.\n"));
                        return TRUE; /* TRUE indicates that the signal was handled. */
                case CTRL_CLOSE_EVENT:
                        _tprintf(_T("Close event received by handler. Leaving the handler
in 5 seconds or less.\n"));
                        exitFlag = TRUE;
                        Sleep(4000); /* Decrease this time to get a different effect
```

```
*/
                        _tprintf(_T("Leaving handler in 1 second or less.\n"));
                        return TRUE; /* Try returning FALSE. Any difference? */
                default:
                        _tprintf(_T("Event: %d received by handler. Leaving in 5 seconds
or less.\n"), cntrlEvent);
                        exitFlag = TRUE;
                        Sleep(4000); /* Decrease this time to get a different effect */
                        _tprintf(_T("Leaving handler in 1 seconds or less.\n"));
                        return TRUE; /* TRUE indicates that the signal was handled. */
        }
}
```

The program can be terminated by the user either by closing the console or with a Ctrl-C. The handler will register with windows using the SetConsoleCtrlHandler(Handler, TRUE) function. The handler will be activated upon the occurrence of any console event. If the registration of any handler fails due to any reason, then an error message will be printed.

Lesson 61

Exception handling functions can be directly associated with an exception just like console handlers. In case a vectored exception is set up, then windows first looks for Vectored Exception Handlers (VEH) and then unwinds the stack.

No __try and __catch keywords are required with VEH and they are like console control handlers. Windows provides a set of APIs for VEH management as follows.

PVOID WINAPI AddVectoredExceptionHandler(ULONG FirstHandler, PVECTORED_EXCEPTION_HANDLER VectoredHandler);

The given API has two parameters; FirstHandler is the parameter used to specify the order in which the handler executes. Non-zero value indicates that it will be the first one to execute and zero specifies it to be the last. If there are more than one handlers setup with zero value, then they will be invoked in the order they are added using AddVectoredExceptionHandler(). Return value is NULL in case of failure, otherwise it returns the Handle to the Vectored Exception Handler.

A call to RemoveVectoredExceptionHandler() is used to remove the vectored exception from a chain. Following is the structure of a vectored exception handler to be set up by the AddVectoredExceptionHandler().

LONG WINAPI VectorHandler(PEXCEPTION_POINTERS ExceptionInfo);

PEXCEPTION_POINTER is a pointer to EXCEPTION_POINTER structure that was previously retrieved by GetExceptionInformation().

Pitfalls of Vectored Exceptions

The exception handler function should be fast and must return as quickly as possible, therefore it should not have a lot of code. The VEH should neither perform any blocking operation like the Sleep() function nor use any synchronization objects. Typically a VEH would access exception information structure, do some minimal processing, and set a few flags.

VEH uses the same return values as SHE.

EXCEPTION_CONTINUE_EXECUTION : No more handlers invoke, exception stack does not unwind, and the execution continues from the point where the exception occurred.

EXCEPTION_CONTINUE_SEARCH : The next VEH handler is probed, if there are no additional VEH handlers, then SEH stack is unwound.

Lesson 62

Dynamic Memory

Need of dynamic memory arises whenever dynamic data structures like search tables, trees, linked lists etc. are used. Windows provides a set of APIs for handling dynamic memory allocation.

Windows also provides memory mapped files which allows direct movement of data to and from user space and files without the use of file APIs. Memory Mapped files can help conveniently

handle dynamic data structure and make file handling faster because they are treated just like memory. It also provides a mechanism for memory sharing among processes.

Windows Memory Management

Windows essentially uses two API platforms i.e. Win32 and Win64.

The Win32 API uses pointers of size 32 bits, hence the virtual space is 2^32. All data types have been optimized for 32 bit boundaries. Win64 uses a virtual space of 2^64 (16 Exabytes).

A good strategy is to design an application in such a way that it could run in both modes without any change in code.

Win32 makes at least half of the virtual space (8GB) accessible to a process and the rest of the space is reserved by the system for shared data, code, and drivers etc. Overall Windows provides a large memory space available to user programs and hence requires optimal management.

Lesson 63

Dynamic Memory

Need of dynamic memory arises whenever dynamic data structures like search tables, trees, linked lists etc. are used. Windows provides a set of APIs for handling dynamic memory allocation.

Windows also provides memory mapped files which allows direct movement of data to and from user space and files without the use of file APIs. Memory Mapped files can help conveniently handle dynamic data structure and make file handling faster because they are treated just like memory. It also provides a mechanism for memory sharing among processes.

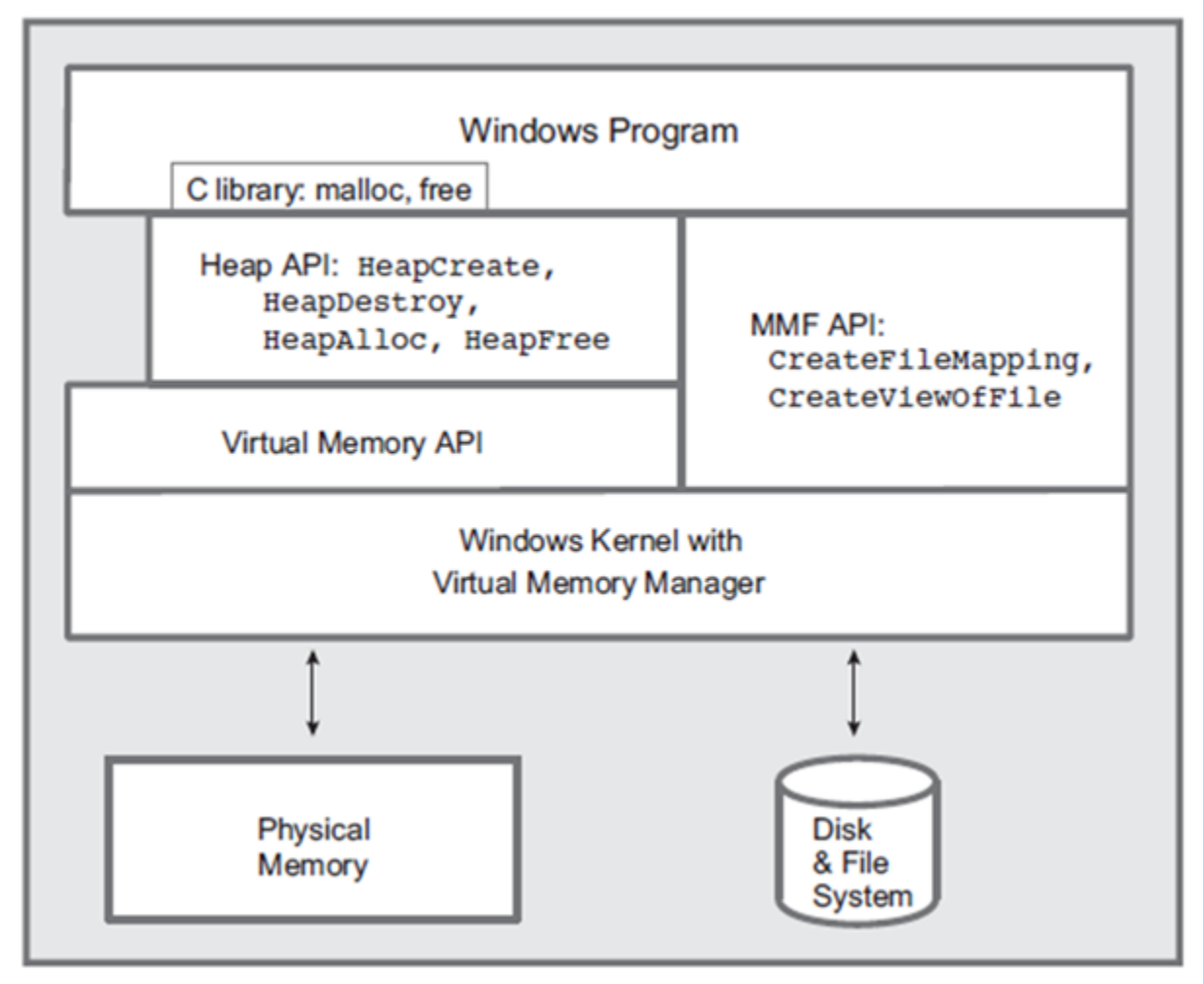
Windows Memory Management

Windows essentially uses two API platforms i.e. Win32 and Win64.

The Win32 API uses pointers of size 32 bits, hence the virtual space is 2^32. All data types have been optimized for 32 bit boundaries. Win64 uses a virtual space of 2^64 (16 Exabytes).

A good strategy is to design an application in such a way that it could run in both modes without any change in code.

Win32 makes at least half of the virtual space (8GB) accessible to a process and the rest of the space is reserved by the system for shared data, code, and drivers etc. Overall Windows provides a large memory space available to user programs and hence requires optimal management.



Further information about the parameters of Windows Memory Management can be probed using the following API.

VOID GetSystemInfo(LPSYSTEM_INFO lpSysInfo)

The API returns a pointer to SYSTEM_INFO structure. The structure contains various information regarding the system like page size, granularity, and application's physical memory address space.

Lesson 64

A programmer allocates memory dynamically from a heap. Windows maintains a pool of heaps and a process can have many heaps. Traditionally, one heap is considered enough. But several heaps may be required to make a program more efficient.

In case a single heap is sufficient, then a runtime library function for heap allocation like malloc(), free(), calloc(), realloc() might be enough.

Heap is a windows object and hence is accessed by a handle. Whenever you require allocating memory from heap, you need a heap handle. Every process in windows has a default heap which can be accessed through the following API.

HANDLE GetProcessHeap(VOID)

The API returns a handle to the process heap. NULL is returned in case of failure and not INVALID_HANDLE_VALUE.

However, due to a number of reasons it would be desirable to have more than one heap. Sometimes it is convenient to have distinct heaps for different data structures.

Separate Heaps

Following are the benefits of separate heaps:

1. If a distinct heap is assigned to each thread, then each thread will only be able to use the memory allocated to each thread.

2. Separation of memory space among threads reduces memory contention.

3. Fragmentation is reduced when one fixed size data structure is allocated from a single heap.

4. Allocating a single heap among each thread simplifies synchronization.

5. If a single heap contains complex data structures, then they can be easily de-allocated with a single API call by de-allocating the entire heap. We will not need complex de-allocation algorithms in such cases.

6. Small heaps for a single data structure reduces the chances of page faults as per the principle of locality.

Lesson 65

We can create a new heap using HeapCreate() API and its size can be set to zero. The API adjusts the heap size to the nearest multiple of page size. Memory is committed to the heap initially, rather than on demand. In case the memory requirements increase than the initial, more pages will automatically be allocated to the heap up to maximum size allowed.

If the required memory is not known, then deferring memory commitment is a good practice as heap is a limited resource. Following is the syntax of the API used to create new heaps.

HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize);

dwMaximumSize if non-zero, determines the maximum limit of the heap memory set by the user. Heap is not grow-able beyond this point. In case it's zero, then the heap is grow-able to the extent of the virtual memory space available for the heap.

dwInitialSize is the initial size of the heap set by the programmer. SIZE_T is used to enable portability. Based on the win32 or win64 platforms, SIZE_T will be 32 or 64 bit wide.

flOptions is usually a combination of the following three flags predefined in Windows:

1. HEAP_GENERATE_EXCEPTIONS: This flag will raise exceptions in case there is any failure while allocating memory to heap. Exceptions are not generated by CreateHeap(), rather they may occur at the time of allocation.

2. HEAP_NO_SERIALIZE: This option provides a slight performance improvement when serialization of data is not required.

3. HEAP_CREATE_ENABLE_EXECUTE: It is an advanced option that allows you to execute code from heap. Usually an exception will occur if data from heap is interpreted as code due to the data execution prevention (DEP) restriction by Windows.

CloseHandle() is not appropriate to dispose of a heap handle rather HeapDestroy() is used as follows.

```
BOOL HeapDestroy(

  HANDLE hHeap

);
```

hHeap is the handle to a previously created heap. Do not use the handle obtained from GetProcessHeap() because it may raise an exception. This is an easy way to get rid of all the contents of the heap including complex data structures.

Lesson 66

Managing Heap Memory

Once a heap is created, it does not allocate memory that is directly available to the program. Rather, it only creates a logical structure of heap that will be used to allocate new memory blocks. Memory blocks are allocated using heap memory allocation APIs like HeapAlloc() and HeapReAlloc().

LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);

hHeap is the handle of the heap from which memory is to be allocated. dwFlags are quite similar to the flags used in HeapCreate().

HEAP_GENERATE_EXCEPTIONS: This flag will raise exceptions in case there is any failure while allocating memory to heap. Exceptions are not generated by CreateHeap(), rather they may occur at the time of allocation.

HEAP_NO_SERIALIZE: This option provides a slight performance improvement when serialization of data is not required.

The first flag HEAP_GENERATE_EXCEPTION is ignored if it is already set in HeapCreate().

HEAP_ZERO_MEMORY: Specifies that the allocated memory will be initialized to zero.

dwBytes is the size of the memory block to be allocated. For non-grow-able heap, its 0x7FFF8 approximately equivalent to 0.5 MB.

The return value of the function is LPVOID. This is the address of the allocated memory block. Use this pointer in a formal way and there is no need to make any reference to the Heap handle.

In case HEAP_GENERATE_EXCEPTION flag is set, an exception is generated if failure occurs. The exception is either STATUS_NO_MEMORY or STATUS_ACCESS_VIOLATION.

If the exception flag is not set, then NULL is returned by HeapAlloc() and the GetLastError() does not work on HeapAlloc().

BOOL HeapFree(HANDLE hHeap, DWORD dwFlags,

_Frees_ptr_opt_ LPVOID lpMem);

hHeap is the heap handle from which memory is to be allocated. dwFlags should be 0 or set to HEAP_NO_SERIALIZE. lpMem should be the pointer previously returned by HeapAlloc() or HeapReAlloc().

Return value of FALSE will indicate a failure. GetLastError() can be used to get the error.

LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags,

_Frees_ptr_opt_ LPVOID lpMem, SIZE_T dwBytes);

hHeap is the heap handle from which memory is to be allocated. dwFlags: HEAP_GENERATE_EXCEPTION and HEAP_NO_SERIALIZE are quite similar to the flags used in HeapAlloc().

HEAP_ZERO_MEMORY: only the newly allocated memory is set to zero (in case dwBytes is greater than the previous allocation).

HEAP_REALLOC_IN_PLACE_ONLY: It indicates the newly allocated block should lie contiguously to the previously allocated block.

lpMem: specifies the pointer to the block previously allocated to the same heap hHeap.

dwBytes: It refers to the block size to be allocated that can be lesser or greater than the previous allocation. But the same restriction as HeapAlloc applies i.e. the block size cannot be greater than 0x7FFF8.

Lesson 67

Some programs may require to determine the size of allocated blocks in the Heap. The size of the allocated block is determined using the API HeapSize() as follows.

```
SIZE_T HeapSize(
  HANDLE  hHeap,
  DWORD   dwFlags,
  LPCVOID lpMem
);
```

The function returns the size of the block or zero in case of failure. The only valid dwFlag is HEAP_NO_SERIALIZE.

Serialization

Serialization is required when dealing with concurrent threads using some common resource. Serialization is not required if threads are autonomous and there is no possibility of concurrent threads disrupting each other.

The flag HEAP_NO_SERIALIZE provides an estimated performance advantage of 16%. Using this flag can improve performance if there are no conflicts. This flag can be conveniently used if:

a. The program is single threaded.

b. Each thread has its own heap that is insulated from other threads.

c. The program has its own mutual exclusion mechanism.

Heap Exceptions

Heap exceptions are enabled using the flag HEAP_GENERATE_EXCEPTION. This allows the program to close open "handles" before a program terminates. There can be two scenarios with this option:

a. STATUS_NO_MEMORY: It means there is no more memory to be allocated because the heap has reached its limits.

b. STATUS_ACCESS_VIOLATION: It indicates that the heap has been corrupted. Either the heap has been accessed as code, or has been accessed beyond its bounds.

There are some other functions that can be used while working with heaps. For example,

HeapSetInformation() can be used to enable low fragmentation mode. It can also be used to allow termination of a thread upon heap's corruption.

HeapCompact() allows finding the largest allocated block in the heap.

HeapValidate() allows detecting heap corruption.

HeapWalk() enumerates the allocated blocks on heap.

HeapLock() and HeapUnLock() allow serializing access to the heap.

Lesson 68

Using Memory Management APIs

Till now we have used the Memory Management APIs for allocating, reallocating and deallocating heaps. We can also get the size of a heap through an API.

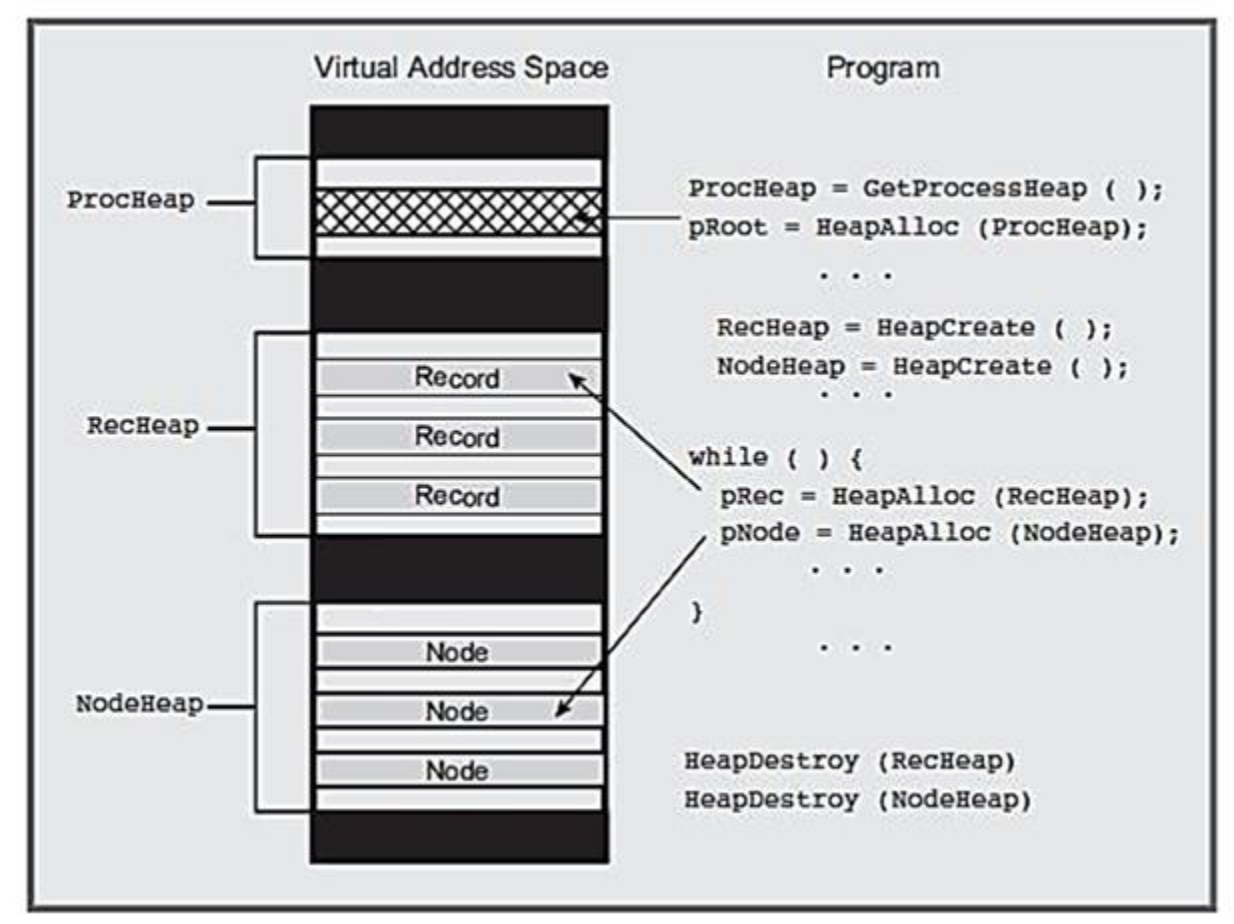
A typical methodology of dealing with heaps should be to get a heap handle either using HeapCreate() or GetProcessHeap(). Use the handle obtained from the above to allocate memory blocks from the heap using HeapAlloc(). If some block needs to be deallocated, use HeapFree(). Before the program is terminated or when the heap is not required, use HeapDestroy() to dispose of the heap.

It is convenient not to mix up windows heap API and Run Time Library functions. Anything allocated with C library functions should also be deallocated with C library functions.

Lesson 69

A Sorting File example using heaps

The example is formulated using two heaps. The first one will be a node heap, while the other is a record heap. Node heap will be used to build a tree, while the data heap will be used to store keys.



Following are the three heaps as shown in the figure:

1. ProcHeap
2. RecHeap
3. NodeHeap

ProcHeap contains the root address, but RecHeap stores the records. NodeHeap on the other hand stores the nodes when they are created. Sorting will be performed in the NodeHeap that gives a reference to be searched in the RecHeap. The data structure will be maintained in the NodeHeap. At the end, all the heaps will be destroyed except ProcHeap because it is created using GetProcessHeap().

Lesson 70

The example is formulated using two heaps. One is a node heap and the other is a data heap. Node heap is used to build a tree, while data heap is used to store keys. Recursive functions are used for allocating nodes and scanning nodes as tree is a recursive structure. Data in file is read in a record and the key is used to lexically build a tree. The tree only contains the key entries. The file is ultimately sorted by an In-order traversal of the tree.

/* Chapter 5, sortBT command. Binary Tree version. */

/* sort files

Sort one or more files.

This limited implementation sorts on the first field only.

The key field length is assumed to be fixed length (8-characters).

The data fields are varying length character strings. */

/* This program illustrates:

1. Multiple independent heaps; one for the sort tree nodes,

the other for the records.

2. Using HeapDestroy to free an entire data structure in a single operation.

3. Structured exception handling to catch memory allocation errors. */

/* Technique:

1. Scan the input file, placing each key (which is fixed size)

into the binary search tree and each record onto the data heap.

2. Traverse the search tree and output the records in order.

3. Destroy the heap and repeat for the next file. */

```
#include "Everything.h"

#define KEY_SIZE 8

		/* Structure definition for a tree node. */

typedef struct _TREENODE {

		struct _TREENODE *Left, *Right;

		TCHAR key[KEY_SIZE];

		LPTSTR pData;

} TREENODE, *LPTNODE, **LPPTNODE;

#define NODE_SIZE sizeof (TREENODE)

#define NODE_HEAP_ISIZE 0x8000

#define DATA_HEAP_ISIZE 0x8000

#define MAX_DATA_LEN 0x1000

#define TKEY_SIZE KEY_SIZE * sizeof (TCHAR)

#define STATUS_FILE_ERROR 0xE0000001	// Customer exception

LPTNODE FillTree (HANDLE, HANDLE, HANDLE);

BOOL Scan (LPTNODE);

int KeyCompare (LPCTSTR, LPCTSTR), iFile; /* for access in exception handler */

BOOL InsertTree (LPPTNODE, LPTNODE);

int _tmain (int argc, LPTSTR argv[])

{

		HANDLE hIn = INVALID_HANDLE_VALUE, hNode = NULL, hData =
NULL;

		LPTNODE pRoot;
```

```
BOOL noPrint;

CHAR errorMessage[256];

int iFirstFile = Options (argc, argv, _T ("n"), &noPrint, NULL);

if (argc <= iFirstFile)

    ReportError (_T ("Usage: sortBT [options] files"), 1, FALSE);

                /* Process all files on the command line. */

for (iFile = iFirstFile; iFile < argc; iFile++) __try {

                        /* Open the input file. */

    hIn = CreateFile (argv[iFile], GENERIC_READ, 0, NULL,
OPEN_EXISTING, 0, NULL);

    if (hIn == INVALID_HANDLE_VALUE)

        RaiseException (STATUS_FILE_ERROR, 0, 0, NULL);


    __try {

        /* Allocate the two growable heaps. */

        hNode = HeapCreate (

    HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE,
    NODE_HEAP_ISIZE, 0);

        hData = HeapCreate (

    HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE,
    DATA_HEAP_ISIZE, 0);

/* Process the input file, creating the tree. */

        pRoot = FillTree (hIn, hNode, hData);

        /* Display the tree in key order. */
```

```
                    if (!noPrint) {
                                _tprintf (_T ("Sorted file: %s\n"), argv[iFile]);
                                Scan (pRoot);
                    }
                } __finally { /* Heaps and file handle are always closed */
                    /* Destroy the two heaps and data structures. */
                    if (hNode != NULL) HeapDestroy (hNode);
                    if (hData != NULL) HeapDestroy (hData);
                    hNode = NULL; hData = NULL;
                    if (hIn != INVALID_HANDLE_VALUE) CloseHandle (hIn);
                    hIn = INVALID_HANDLE_VALUE;
                }
            } /* End of main file processing loop and try block. */
            /* Handle the exceptions that we can expect - Namely, file open error or out of memory. */
            __except ( (GetExceptionCode() == STATUS_FILE_ERROR ||
GetExceptionCode() == STATUS_NO_MEMORY)
                                ? EXCEPTION_EXECUTE_HANDLER :
EXCEPTION_CONTINUE_SEARCH)
            {
                _stprintf (errorMessage, _T("\n%s %s"), _T("sortBT error on file:"),
argv[iFile]);
                ReportError (errorMessage, 0, TRUE);
            }
            return 0;
```

```
}
LPTNODE FillTree (HANDLE hIn, HANDLE hNode, HANDLE hData)
/* Scan the input file, creating a binary search tree in the
                hNode heap with data pointers to the hData heap. */
/* Use the calling program's exception handler. */
{
                LPTNODE pRoot = NULL, pNode;
                DWORD nRead, i;
                BOOL atCR;
                TCHAR dataHold[MAX_DATA_LEN];
                LPTSTR pString;
                                                /* Open the input file. */
while (TRUE) {
                        pNode = HeapAlloc (hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
                        pNode->pData = NULL;
                        (pNode->Left) = pNode->Right = NULL;
                                                /* Read the key. Return if done. */
                        if (!ReadFile (hIn, pNode->key, TKEY_SIZE,
                                                &nRead, NULL) || nRead != TKEY_SIZE)
                                                /* Assume end of file on error. All records
must be just the right size */
                                return pRoot;   /* Read the data until the end of line. */
                        atCR = FALSE;           /* Last character was not a CR. */
```

```
                    for (i = 0; i < MAX_DATA_LEN; i++) {
                            ReadFile (hIn, &dataHold[i], TSIZE, &nRead, NULL);
                            if (atCR && dataHold[i] == LF) break;
                            atCR = (dataHold[i] == CR);
}
                    dataHold[i - 1] = _T('\0');
                    /* dataHold contains the data without the key.
                            Combine the key and the Data. */
                    pString = HeapAlloc (hData, HEAP_ZERO_MEMORY,
                                            (SIZE_T)(KEY_SIZE + _tcslen (dataHold) + 1) *
TSIZE);
                    memcpy (pString, pNode->key, TKEY_SIZE);
                    pString[KEY_SIZE] = _T('\0');
                    _tcscat (pString, dataHold);
                    pNode->pData = pString;
                                            /* Insert the new node into the search tree. */
                    InsertTree (&pRoot, pNode);
        } /* End of while (TRUE) loop */
                return NULL; /* Failure */
}
BOOL InsertTree (LPPTNODE ppRoot, LPTNODE pNode)
/* Insert the new node, pNode, into the binary search tree, pRoot. */
{
```

```
        if (*ppRoot == NULL) {
                *ppRoot = pNode;
                return TRUE;
        }
        if (KeyCompare (pNode->key, (*ppRoot)->key) < 0)
                InsertTree (&((*ppRoot)->Left), pNode);
        Else
    InsertTree (&((*ppRoot)->Right), pNode);
        return TRUE;
}
int KeyCompare (LPCTSTR pKey1, LPCTSTR pKey2)
/* Compare two records of generic characters.
        The key position and length are global variables. */
{
        return _tcsncmp (pKey1, pKey2, KEY_SIZE);
}
static BOOL Scan (LPTNODE pNode)
/* Scan and print the contents of a binary tree. */
{
        if (pNode == NULL)
                return TRUE;
        Scan (pNode->Left);
        _tprintf (_T ("%s\n"), pNode->pData);
```

```
        Scan (pNode->Right);

        return TRUE;

}
```

Lesson 71

Memory Mapping

Dynamic memory is allocated from the paging file. The paging file is controlled by the Operating System's (OS) virtual memory management system. Also the OS controls the mapping of virtual address onto physical memory. Memory mapped files help to directly map virtual memory space onto a normal file.

Advantages of Memory Mapped IO

There is no need to invoke direct file Input Output (IO) operations. Any data structure placed in file will be available for later use as well. It is convenient and efficient to use in-memory algorithms for sorting, searching etc. Large files could be processed as if they are placed in memory. File processing is faster than ReadFile() and WriteFile(). There is no need to manage buffers for repetitive operation on a file. This is more optimally done by OS. Multiple processes can share memory space by mapping their virtual memory space onto a file. For file operations, page file space is not needed.

Other considerations

Windows also use memory mapping while implementing Dynamic Link Libraries (DLLs) and loading & executing executable (EXE) files. It is strongly recommended to use SHE exception handling while dealing with memory mapped file to look for EXCEPTION_IN_PAGE_ERROR exceptions.

Lesson 72

File Mapping Objects

In order to perform memory mapped file IO operations, file mapping objects need to be created. This object uses the file handle of an open file. The open file or part of the file is mapped onto the address space of the process. File mapping objects are assigned names so that they are also available to other processes. Moreover, these mapping objects also require protection and security attributes and a size. The API used for this purpose is CreateFileMapping().

HANDLE CreateFileMapping(HANDLE hFile,

LPSECURITY_ATTRIBUTES lpFileMappingAttributes,

DWORD flProtect,

DWORD dwMaximumSizeHigh,

DWORD dwMaximumSizeLow, LPCTSTR lpName);

hFile is the open handle to file compatible with protection flag dwProtect

INVALID_HANDLE_VALUE refers to the paging file

LPSECURITY_ATTRIBUTES allow the mapping object to be secured

dwProtect specifies the mapping file access with following attributes

PAGE_READONLY - It means that page can only be read within the mapped region. It can neither be written nor executed. hFile must have GENERIC_READ access.

PAGE_READWRITE - Provides full access to object if the hFile has GENERIC_READ and GENERIC_WRITE access.

PAGE_WRITECOPY - Means that when a mapped region changes, a private copy is written to the paging file and not to the original file.

dwMaximumSizeHigh and dwMaximumSizeLow specify the size of the mapping object. If set to 0, then the current file size is used. Carefully specify this size in the following cases:

· If the file size is expected to grow, then use the expected file size.

· Do not map a region beyond this limit. Once the size is assigned the mapping region cannot grow.

· An attempt to create a mapping on a zero length file will fail.

· The mapping size needs to be specified in the form of two 32-bit values rather than one 64-bit value.

· lpMapName is the name of the map that can also be used by other processes. Set this to NULL if you do not mean to share the map.

Lesson 73

Open Existing File Mapping Objects

Previously, we discussed that a file mapping can be assigned a shared name by using CreateFileMapping(). This shared name can be used to open existing file maps using OpenFileMapping(). A file map created by a certain process can be subsequently used by other processes by referring to the object by name. The operation may fail if the name does not exist.

HANDLE OpenFileMapping(DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCSTR lpMapName);

dwDesiredAccess is checked against the accessed assigned by CreateFileMapping();

lpMapName is the name of the mapping assigned by CreateFileMapping().

Lesson 74

Mapping into Process Space

Previously, file mapping was created or a handle to already created mapping handle was obtained. The next step is to assign a process address space to file mapping. In case of heaps, first heap was created and then HeapAlloc() was used to allocate space within heap. Similarly once the file mapping is created, MapViewOfFile() is used to define file view block.

LPVOID MapViewOfFile(HANDLE hFileMappingObject,

DWORD dwDesiredAccess,

DWORD dwFileOffsetHigh,

DWORD dwFileOffsetLow,

SIZE_T dwNumberOfBytesToMap);

hFileMappingObject is file mapping object previously obtained by CreateFileMapping().

dwDesiredAccess should be compatible with access rights of file mapping object. The three flag commonly used are:

FILE_MAP_WRITE

FILE_MAP_READ

FILE_MAP_ALL_ACCESS

dwFileOffsetHigh and dwFileOffsetLow give the starting address of the file from where the mapping starts. To start the mapping from the start of a file, set both as zero. This value should be specified in multiples of 64K.

dwNumberOfBytesToMap shows the number of bytes of file to map. Set as zero to map the whole file.

If the function is successful it returns the base address of the mapped region. If the function fails, the return value is NULL.

Closing Mapping Handle

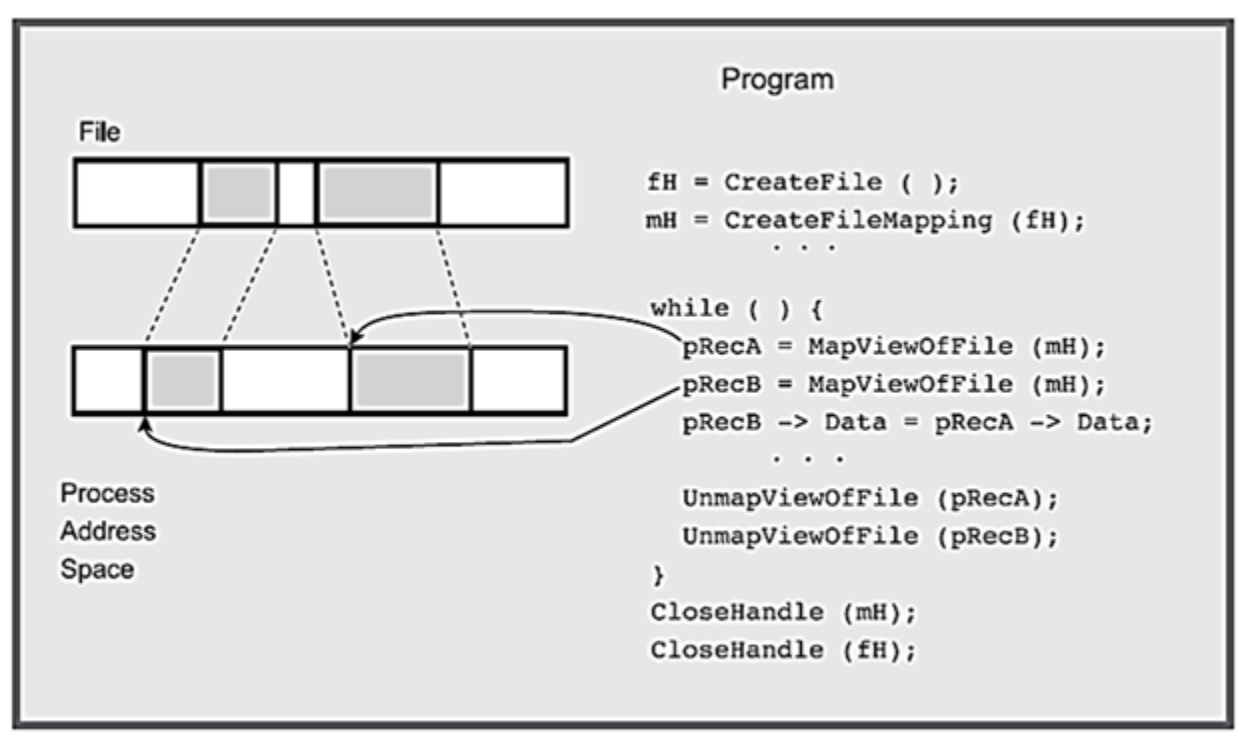
As it is necessary to release Heap blocks with HeapFree(), it is also necessary to unmap file views. File views are unmapped using UnmapViewOfFile().

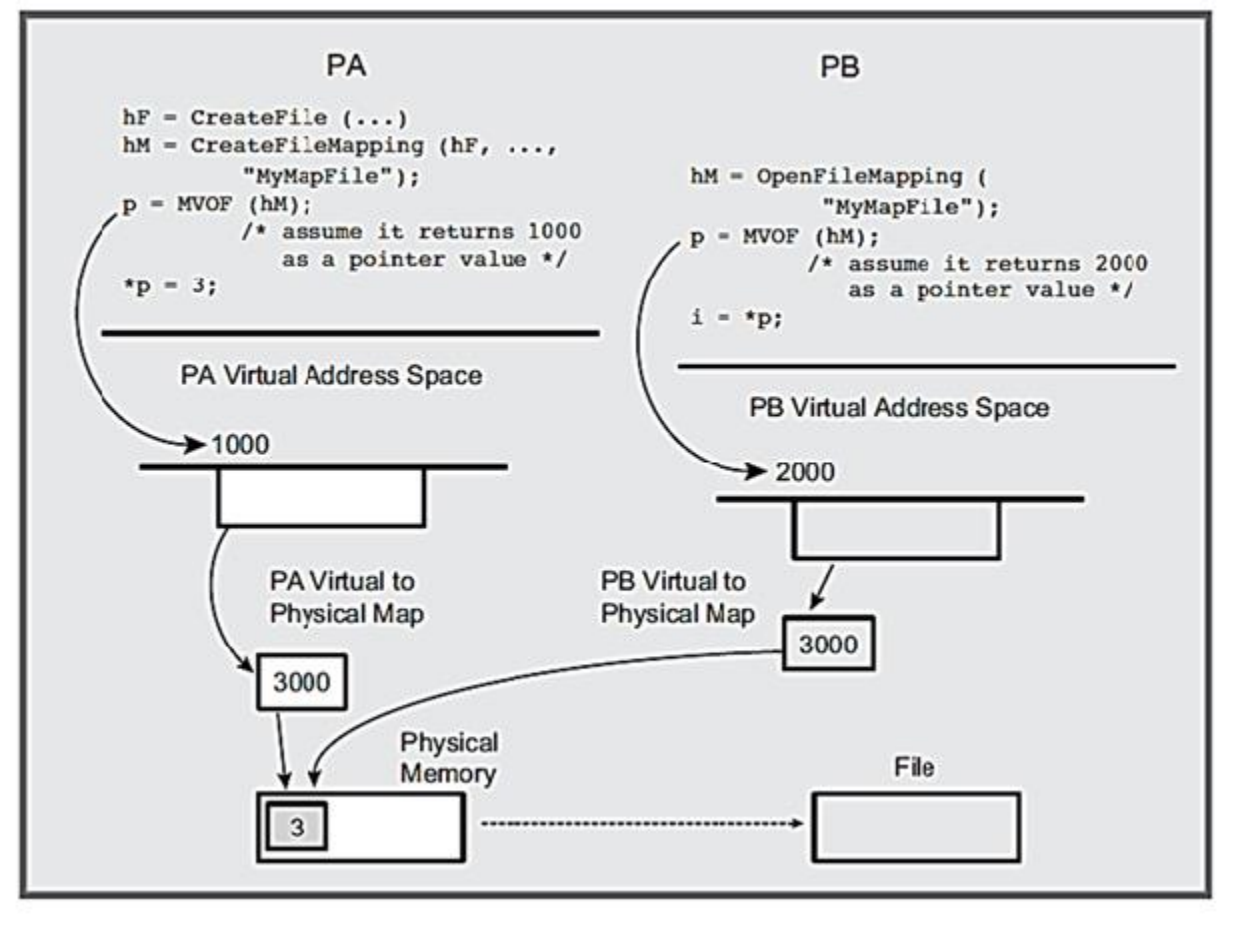
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);

lpBaseAddress is the pointer to the base address of the mapped view. If the function fails, the return value is zero.

Flushing File View

The file view can be flushed using the FlushViewOfFile() API. This will force the OS to writeback the dirty pages of the file on to disk. In case two processes try to access a file at a time such that one uses file mapping and other uses ReadFile() and WriteFile(). Then both processes may not receive the same view. Changes made through file maps might still be in memory and may not be accessible through ReadFile or WriteFile unless they are flushed. To get a uniform view, it is necessary that all the processes use file maps.





Lesson 75

File Mapping Limitations

In Win32, it is not possible to map files bigger than 2-3 GB. Also the entire 3GB might not be available for merely file space. The above limitation is alleviated in Win64. File mapping cannot be extended. You need to know the size of a map in advance. Customized functions would be required to allocate memory within the mapped region.

File Mapping Procedure

The following minimum steps need to be taken while working with mapped files:

· Open File with at least GENERIC_READ access.

· If the file is new then set its length as some non-zero value using SetFilePointerEx() followed by SetEndOfFile().

· Map the file using CreateFileMapping() or OpenFileMapping ().

· Create one or more map views using MapViewOfFile().

· Access file through memory references.

· Unmap and then again Map file view to change regions.

· Use SEH to catch EXCEPTIO_IN_PAGE_ERROR exceptions.

· In the end, unmap file view with UnmapViewOfFile() and use CloseHandle() to close map and file handles.

Lesson 76

Sequential File Access

Accessing a file through file mapping presents visible advantages. Although the setting up of fileviews might be programmatically complex, the advantages are far bigger. The processing time may reduce 3 folds as compared to conventional file operations while dealing with sequential files. These advantages may only seem to disappear if the size of input and output files is too large. The example is a simple Ceasar cipher application. It sequentially processes all the characters with the file. It simply substitutes each character by shifting it a few places in the ASCII set.

/* Chapter 5.

cci_fMM.c function: Memory Mapped implementation of the

simple Caeser cipher function. */

#include "Everything.h"

BOOL cci_f (LPCTSTR fIn, LPCTSTR fOut, DWORD shift)

```
/* Caesar cipher function.

*       fIn:                    Source file pathname.

*       fOut:                   Destination file pathname.

*       shift:                  Numeric shift value */

{

BOOL complete = FALSE;

                HANDLE hIn = INVALID_HANDLE_VALUE, hOut =
INVALID_HANDLE_VALUE;

                HANDLE hInMap = NULL, hOutMap = NULL;

                LPTSTR pIn = NULL, pInFile = NULL, pOut = NULL, pOutFile = NULL;

                __try {

                        LARGE_INTEGER fileSize;

                        /* Open the input file. */

                        hIn = CreateFile (fIn, GENERIC_READ, 0, NULL,

                                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

                        if (hIn == INVALID_HANDLE_VALUE)

                                ReportException (_T ("Failure opening input file."), 1);

/* Get the input file size. */

                        if (!GetFileSizeEx (hIn, &fileSize))

                                ReportException (_T ("Failure getting file size."), 4);

                        /* This is a necessar, but NOT sufficient, test for mappability on 32-bit
systems S

                         * Also see the long comment a few lines below */

                        if (fileSize.HighPart > 0 && sizeof(SIZE_T) == 4)
```

```
                ReportException (_T ("This file is too large to map on a Win32
system."), 4);

                /* Create a file mapping object on the input file. Use the file size. */

                hInMap = CreateFileMapping (hIn, NULL, PAGE_READONLY, 0, 0,
NULL);

                if (hInMap == NULL)

                        ReportException (_T ("Failure Creating input map."), 2);

/* Map the input file */

                /* Comment: This may fail for large files, especially on 32-bit systems

                 * where you have, at most, 3 GB to work with (of course, you have much
less

                 * in reality, and you need to map two files.

                 * This program works by mapping the input and output files in their
entirety.

                 * You could enhance this program by mapping one block at a time for
each file,

                 * much as blocks are used in the ReadFile/WriteFile implementations.
This would

                 * allow you to deal with very large files on 32-bit systems. I have not
taken

                 * this step and leave it as an exercise.

                 */

                pInFile = MapViewOfFile (hInMap, FILE_MAP_READ, 0, 0, 0);

                if (pInFile == NULL)

                        ReportException (_T ("Failure Mapping input file."), 3);

/*  Create/Open the output file. */
```

```
/* The output file MUST have Read/Write access for the mapping to
succeed. */

hOut = CreateFile (fOut, GENERIC_READ | GENERIC_WRITE,

0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);

if (hOut == INVALID_HANDLE_VALUE) {

complete = TRUE; /* Do not delete an existing file. */

ReportException (_T ("Failure Opening output file."), 5);

}

/* Map the output file. CreateFileMapping will expand

the file if it is smaller than the mapping. */

hOutMap = CreateFileMapping (hOut, NULL, PAGE_READWRITE,
fileSize.HighPart, fileSize.LowPart, NULL);

if (hOutMap == NULL)

ReportException (_T ("Failure creating output map."), 7);

pOutFile = MapViewOfFile (hOutMap, FILE_MAP_WRITE, 0, 0,
(SIZE_T)fileSize.QuadPart);

if (pOutFile == NULL)

ReportException (_T ("Failure mapping output file."), 8);

/* Now move the input file to the output file, doing all the work in
memory. */

__try

{

CHAR cShift = (CHAR)shift;

pIn = pInFile;
```

```
                pOut = pOutFile;

                while (pIn < pInFile + fileSize.QuadPart) {

                        *pOut = (*pIn + cShift);

                        pIn++; pOut++;

}

                complete = TRUE;

        }

        __except(GetExceptionCode() == EXCEPTION_IN_PAGE_ERROR ?
EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)

        {

                complete = FALSE;

                ReportException(_T("Fatal Error accessing mapped file."), 9);

        }

        /* Close all views and handles. */

        UnmapViewOfFile (pOutFile); UnmapViewOfFile (pInFile);

        CloseHandle (hOutMap); CloseHandle (hInMap);

        CloseHandle (hIn); CloseHandle (hOut);

        return complete;


}

    __except (EXCEPTION_EXECUTE_HANDLER) {

        if (pOutFile != NULL) UnmapViewOfFile (pOutFile); if (pInFile !=
NULL) UnmapViewOfFile (pInFile);
```

```
                    if (hOutMap != NULL) CloseHandle (hOutMap); if (hInMap != NULL)
CloseHandle (hInMap);

                    if (hIn != INVALID_HANDLE_VALUE) CloseHandle (hIn); if (hOut !=
INVALID_HANDLE_VALUE) CloseHandle (hOut);

                    /* Delete the output file if the operation did not complete successfully. */

                    if (!complete)

                        DeleteFile (fOut);

                    return FALSE;

                }

}
```

Lesson 77

Topic 77: Sorting a memory mapped file

Content:

Another advantage of memory mapping is the ability to use convenient memory based algorithms to process files. Sorting data in memory, for instance, is much easier than sorting records in a file.

Program explained in topic 77 sorts a file with fixed-length records. This program, called sortFL, is similar to Program explaining example of sorting with binary search tree that it assumes an 8-byte sort key at the start of the record, but this example is restricted to fixed records.

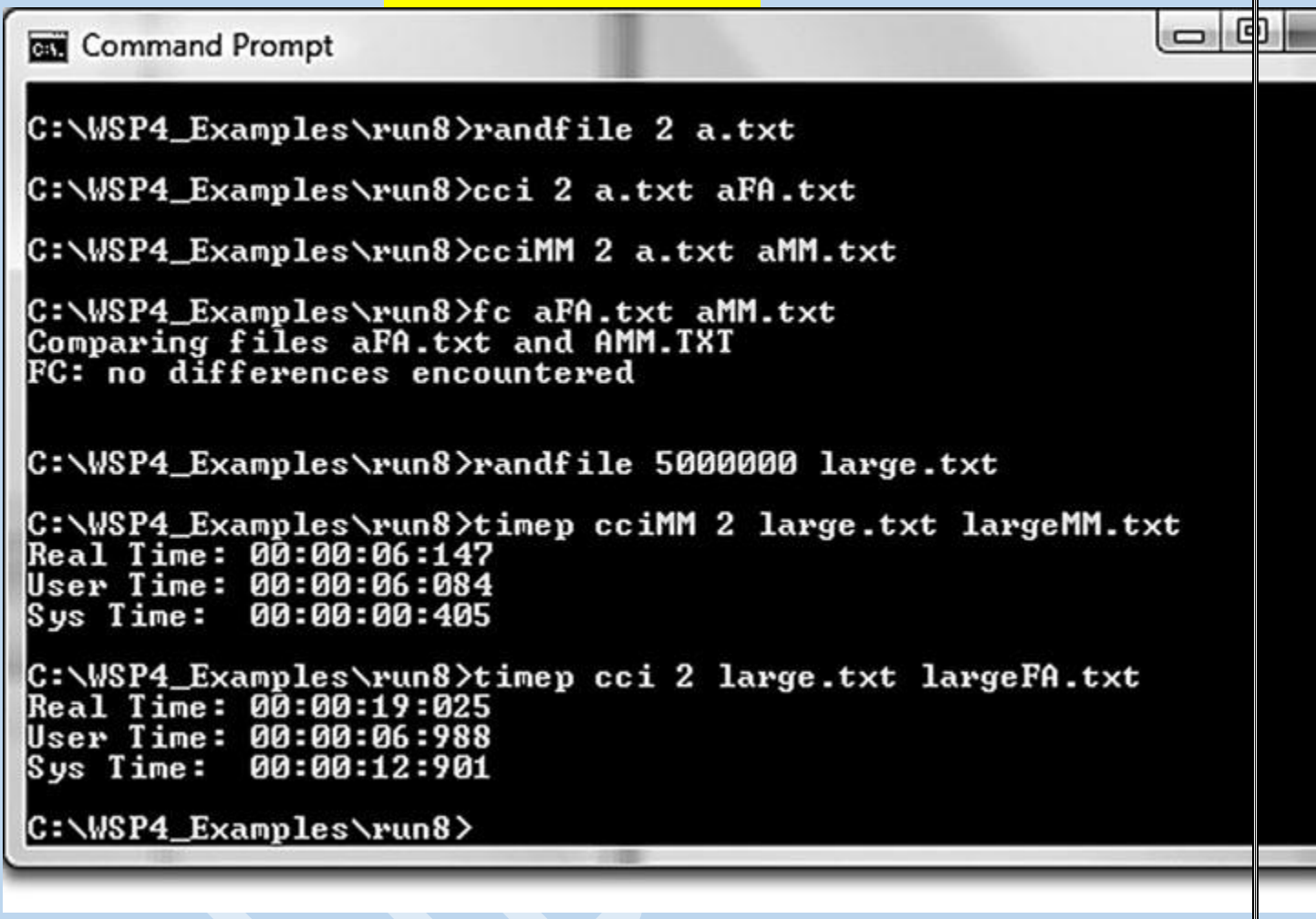


Figure T77-1: File Conversion with Memory-Mapped Files

The sorting is performed by the ***<stdlib.h>*** C library function ***qsort***. Notice that ***qsort*** requires a programmer-defined record comparison function.

This program structure is straightforward. Simply create the file mapping on a temporary copy of the input file, create a single view of the file, and invoke qsort. There is no file I/O. Then the sorted file is sent to standard output using ***_tprintf***, although a null character is appended to the file map.

Exception and error handling are omitted in the listing but are in the Examples solution on the recommended book's Website.

Example of topic 77 is much faster, requiring about 3 seconds to sort a 1,000,000 record file, rather than over 2 minutes.

Program `sortFL`: Sorting a File with Memory Mapping

```
/* Chapter 5. sortFL. File sorting. Fixed-length records. */
/* Usage: sortFL file */

#include "Everything.h"
typedef struct _RECORD {
   TCHAR key[KEY_SIZE];
   TCHAR data[DATALEN];
} RECORD;

#define RECSIZE sizeof (RECORD)

int _tmain (int argc, LPTSTR argv[])
{
   HANDLE hFile = INVALID_HANDLE_VALUE, hMap = NULL;
   LPVOID pFile = NULL;
   LARGE_INTEGER fileSize;
   TCHAR tempFile[MAX_PATH];
   LPTSTR pTFile;

   /* Create the name for a temporary file to hold a copy of
      the file to be sorted. Sorting is done in the temp file.
   _stprintf (tempFile, _T ("%s.tmp"), argv[1]);
   CopyFile (argv[1], tempFile, TRUE);
```

```
    /* Map the temporary file and sort it in memory. */
    hFile = CreateFile (tempFile, GENERIC_READ | GENERIC_WRITE,
          0, NULL, OPEN_EXISTING, 0, NULL);
    GetFileSizeEx (hFile, &fileSize);
    fileSize.QuadPart += 2;
    hMap = CreateFileMapping (hFile, NULL, PAGE_READWRITE,
          fileSize.HighPart, fileSize.LowPart, NULL);
    pFile = MapViewOfFile (hMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);

    qsort (pFile, FsLow / RECSIZE, RECSIZE, KeyCompare);
                          /* KeyCompare is as in Program 5–2. */

    /* Print the sorted file. */
    pTFile = (LPTSTR) pFile;
    pTFile[fileSize.QuadPart/TSIZE] = '\0';
    _tprintf (_T ("%s"), pFile);
    UnmapViewOfFile (pFile);
    CloseHandle (hMap);
    CloseHandle (hFile);
    DeleteFile (tempFile);
    return 0;
}
```

This implementation is straightforward, but there is an alternative that does not require mapping. Just allocate memory, read the complete file, sort it in memory, and write it.

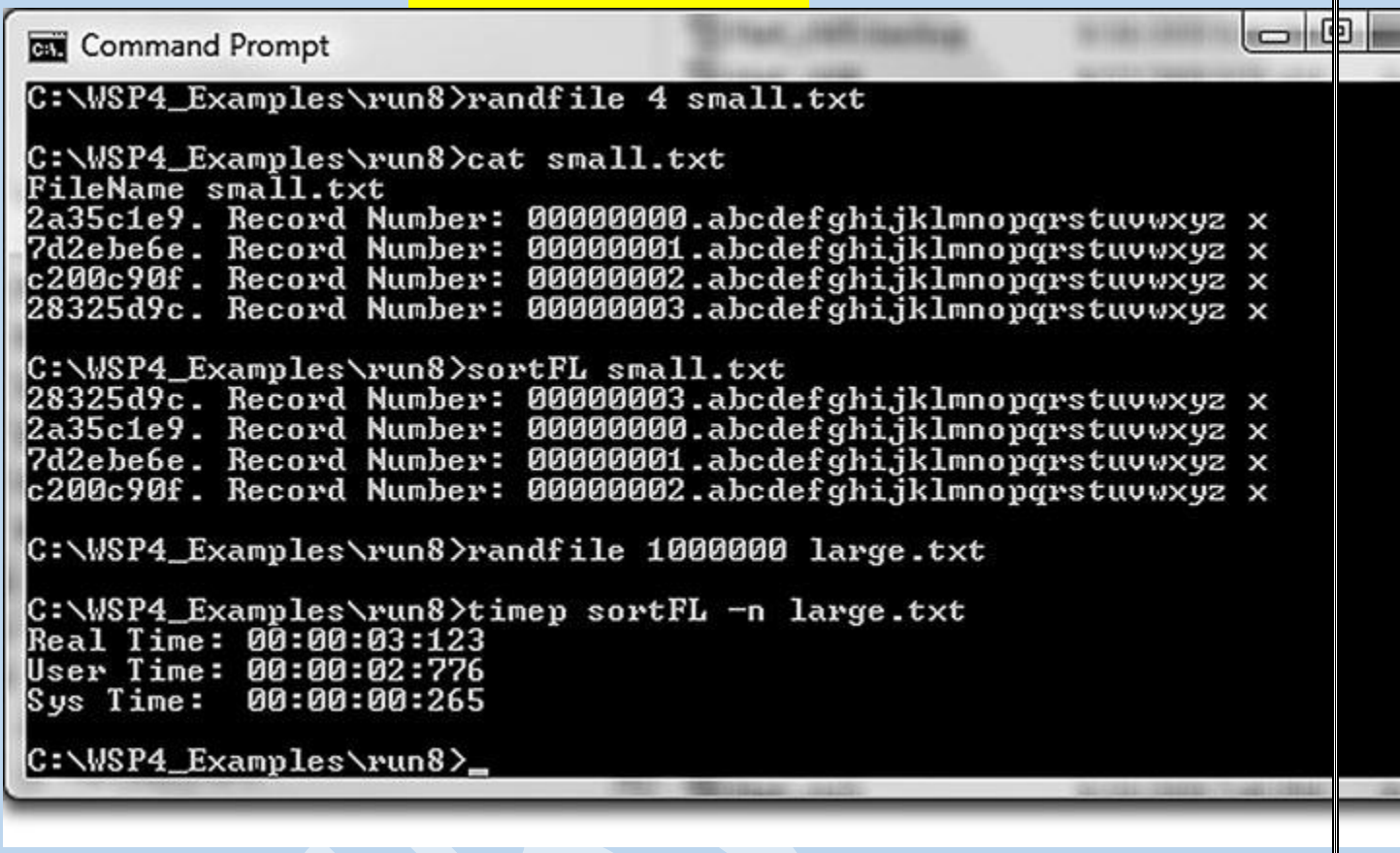


Figure T77-2: Sorting in Memory with File Mapping

Lesson 78

Topic 78: Using Based Pointers

Content:

File maps are convenient, as the preceding examples demonstrate. Suppose, however, that the program creates a data structure with pointers in a mapped file and expects to access that file in the future. Pointers will all be relative to the virtual address returned from MapViewOfFile, and they will be meaningless when mapping the file the next time. The solution is to use based pointers, which are actually offsets relative to another pointer. The Microsoft C syntax, available in Visual C++ and some other systems, is:

type _based (base) declarator

Here are two examples.

```
LPTSTR pInFile = NULL;

DWORD _based (pInFile) *pSize;

TCHAR _based (pInFile) *pIn;
```

Notice that the syntax forces use of the *, a practice that is contrary to Windows convention but which the programmer could easily fix with a typedef.

Lesson 79

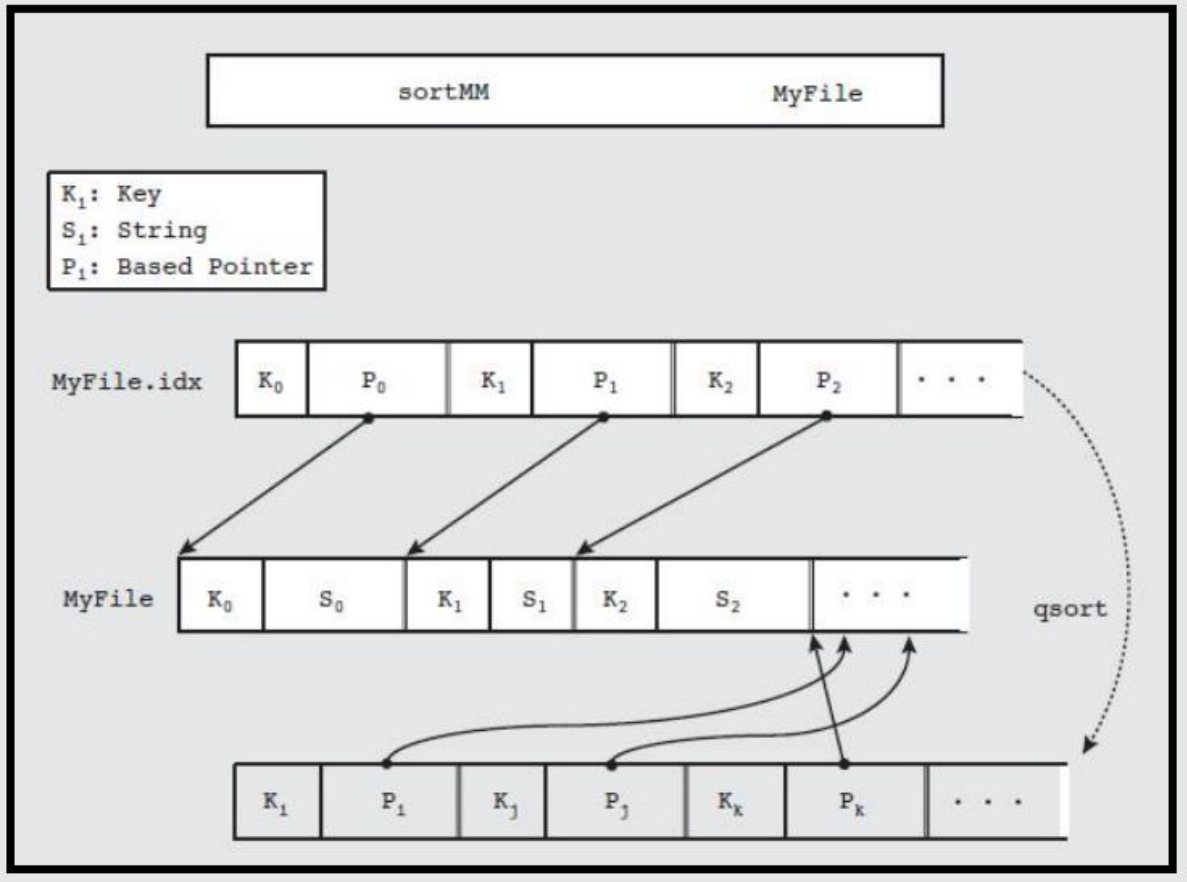
Topic 79: **Based Pointers Example**

Previously we developed a program that sorted record stored in memory mapped file. Sorting huge files each time they are accessed is not feasible, so permanent indexes are maintained on required key.

Using the address returned by MapViewOfFile() for maintaining indexes is meaningless as the address is liable to change in each call to API. A simple methodology is to maintain an array of record and then build an index for the records. And subsequently use the index to access records directly.

Following example program implements this concept in the given way:

The program uses record of varying sizes in a file. It uses the first field of each record as the key of 8 characters. There are two file mapping. One mapping maps the original file and the other maps the index file. Each record in index file contains a key and the pointer location into the original file for the record containing that key. Once index file is created it can be easily used later. Subsequently, index file records can be sorted for faster searching. The input file remains unchanged. Pictorial Representation of the example is attached below;



Lesson 80

Topic 80: Dynamic Link Libraries

We have previously seen the example use of memory mapped files in windows. This a fundamental feature of windows. Windows itself uses this feature while working with Dynamic Link Libraries (DLLs). DLLs are one of the most important components of windows on which many high-level technologies depend like COM.

Static Linking

The conventional approach is to gather all the source code and library functions attach them and encapsulate them into a single executable file. This approach is simple but has few disadvantages.

- The executable image will be large as it contains all library functions. Hence it will consume more disk space and will require large physical memory to run.
- If a library function updates the whole program will require recompilation.

- There can be many programs that require a library function. Each program will have static copy of its own. Hence, resources requirement will increase.
- It will reduced portability as a program compiled with certain environment setting will run same functions in different environment where some other version might be.

How DLLs Solve this Problem

Using DLLs the library functions are not linked at compile time. They are linked at program load time (implicit linking) or at run time (explicit linking).

As a result the size of executable package is smaller. DLLs can be easily used to create shared libraries which can be used by multiple programs concurrently. Only a single copy of shared DLLs is placed in memory. All the processes sharing the DLL map the DLL space onto their program space. Each program will have its own copy of DLL global variables.

New versions or updates can be simply supported by just providing a new DLL without the need of recompiling main code. The library runs in the same processes as the calling the program.

Importance of DLLs

DLLs are used in almost all modern operating systems. DLLs are most important in case of windows as they are used to implement OS interfaces. The entire Windows API is supported by a set of DLLs which are invoked to call kernel services. The DLL code can be shared by multiple processes. DLL function when invoked by a process runs in process space. Therefore, it can use resources of the calling process such as file handles and thread stack. DLLs should be written in Thread-safe manner. A DLL exports variables as well as function entry points.

Lesson 81

Topic 81: Implicit Linking

Implicit linking is the easier of the two techniques. Functions defined in a DLL are collected and build as DLL. The build process builds a .LIB file which is a stub for actual code. The stub is linked to the calling program at build time. It provides a place holder for each function in the DLL.

The place holder/stub will call the original function in the DLL. This file should be placed in common user library directory for the project. The build process also constructs the DLL that contain the original binary image for the functions. This File is usually placed in the same directory as the application.

Function interfaces defined in DLLs should be exported carefully.

Lesson 82

Topic 82: Exporting and Importing Interfaces

For a DLL function to be useful for an application exporting it, it must be declared as exportable. This can be done by using .DEF file or by using __declspec (dllexport) storage modifier.

__declspec (dllexport)DWORD MyFunction (…)

The build process will generate a .LIB file and .DLL file. The .LIB files will a stub for function calls while DLL file will hold the actual code. Similarly the calling program should use the __declspec (dllimport) Storage modifier.

Following Macro can be used with the library file code

#if defined (MYLIBRARY_EXPORTS)

#define LIBSPEC _declspec(dllexport)

#elif defined (__cplusplus)

#define LIBSPEC extern “C” __declspec(dllimport)

#else

#define LIBSPEC __declspec (dllimport)

#endif

LIBSPEC SWORD MyFunction (…)

If you are using Visual C++ compiler then this task is automatically performed by the compiler. When building the calling program you need to specify the .LIB file. When executing the calling program the .DLL file must be placed in the same directory as the application. Following is the default DLL search safe order for explicit and implicit linking.

- Directory containing the application
- The system directory. This path can be determined by GetSystemDirectory()
- The windows directory (GetWindowsDirectory())
- The current directory
- Directories specified by the PATH environment variable

Lesson 83

Topic 83: **Explicit Linking**

Explicit linking requires the program to explicitly a specific DLL to be loaded or freed. Once the DLL is loaded then the program obtains the address of the specific entry point and uses that address as a pointer in function call.

The function is not declared in the calling program. Rather a pointer to a function is declared. The functions required are:

HMODULE LoadLibrary(LPCTSTR lpLibFileName);

HMODULE LoadLibraryEx(LPCTSTR lpLibFileName,

HANDLE hFile,

DWORD dwFlags);

HMODULE is NULL in case of failure.

File name need not mention the extension. The file path must be valid. See MSDN for details of dwFlags Once the DLL is loaded. The programmer needs to obtain an entry point (procedure address) into the DLL. This is done using:

FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);

hModule is the module handle obtained for the DLL. lpProcName cannot b UNICODE. It is the name of the function whose entry point is to be obtained. Its return type is FARPROC. This is the far address of the function. A C type function pointer declaration can be easily used to invoke the function in DLL and pass its parameters.

Lesson 84

Topic 84: **Explicit Linking a file conversion function**

Previously, we have studied many file conversion functions. Some used memory mapped IO, some used file operations. Some performed file conversion some performed file encryption.

Now, we take a look at how we can encapsulate these functions in a DLL and invoke them explicitly.

Following example shows explicit linking of these functions.

```
HMODULE hDLL;

FARPROC pcci;

TCHAR YNResp[5] = YES;

if (argc < 5)

ReportError (_T("Usage: cciEL shift file1 file2 DllName"), 1, FALSE);

 /* Load the cipher function. */

hDLL = LoadLibrary (argv[4]);

 if (hDLL == NULL)

ReportError (_T("Failed loading DLL."), 4, TRUE);

 /* Get the entry point address. */

 pcci = GetProcAddress (hDLL, _T("cci_f"));

if (pcci == NULL)

ReportError (_T ("Failed of find entry point."), 5, TRUE);

cci_f = (BOOL (__cdecl *)(LPCTSTR, LPCTSTR, DWORD)) pcci;

 /* Call the function. */
```

Lesson 85

Topic 85: **DLL Entry Point**

DLL Entry point can be specified optionally when you create a DLL. The code at entry point executes whenever a process attaches to the DLL. In case of implicit linking the DLL attaches at the time of process start and detaches when process ends. In case of explicit linking the process attaches when LoadLibrary()/LoadLibraryEx() is invoked. Also the process detaches when FreeLibrary() is called.

LoadLibraryEx() can also suppress the execution of entry point. Further, entry point is also invoked whenever a thread attaches to the DLL.

The Entry point in DLL is defined as:

BOOL DllMain(

HINSTANCE hDll,

DWORD Reason,

LPVOID lpReserved)

hDll corresponds to the handle returned by LoadLibrary(). lpReserved if NULL, represent explicit attachment else it represents implicit attachment. Reason will have on the four values

DLL_PROCESS_ATTACH

DLL_THREAD_ATTACH

DLL_THREAD_DETACH

DLL_PROCESS_DETACH

Based on the value of Reason the programmer can decide whether to initialize or free resources. All the calls to DllMain are serialized by the system. Serialization is critically important as DllMain() is supposed to perform initialization operations for each thread. There should be no blocking calls, IO calls or wait functions as they will indefinitely block other threads. A call to other DLLs from DllMain() cannot be performed except for few exceptions. LoadLibrary() and LoadLibraryEx() should never be called from DllMain() as it will create more DLL entry points. DisableThreadLibraryCalls() can be used to disable thread attach/detach calls for a specified instance.

Lesson 86

Topic 86: **DLL Version Management**

Complications

Many times a DLL is upgraded to provide more features and new symbols. Also, multiple processes share a single implementation of DLL. This strength of DLL may also lead to some complications.

- If the new DLL has changed interfaces this render problems for older programs that have not been updated for newer version.
- Application that require newer updated functionality may sometime link with older DLL version.

Version Management

One intuitive resolution to the problem can be by using different directory for each version. But there are several solutions. Use DLL version number as part of the .DLL and .LIB file names eg. Utils_4_0.DLL and Utils_4_0.LIB. Applications requiring DLLs can determine the version requirements and subsequently access files with distinct filename using implicit or explicit linking.

Microsoft has introduced a concept of side by side DLLs or assemblies. This solution requires application to declare its DLL requirements using XML.

Querying DLL Information

Microsoft DLLs support a callback function that version information and more regarding a DLL. This callback function can be used dynamically to query the information regarding DLL. The works as follows:

HRESULT CALLBACK DllGetVersion(

DLLVERSION *pdvi

)

Information regarding the DLL is placed in the DLLVERSION structure. It contains a field cbSize which is the size of the structure,

dwMajorVersion

dwMinorVersion

dwBuilderNumber

dwPlatformID

dwPlatformID can be set of DLLVER_PLATFORM_NT if DLL cannot run on windows 9x or to DLLVER_PLATFORM_WINDOWS if there are no restrictions. cbSize should be set to sizeof(DLLVERSION)

Lesson 87

Topic 87: Windows Processes and Threads

Windows Process Management is all about:

- Multitasking Systems
- Multiprocessor systems
- Thread as a unit of execution

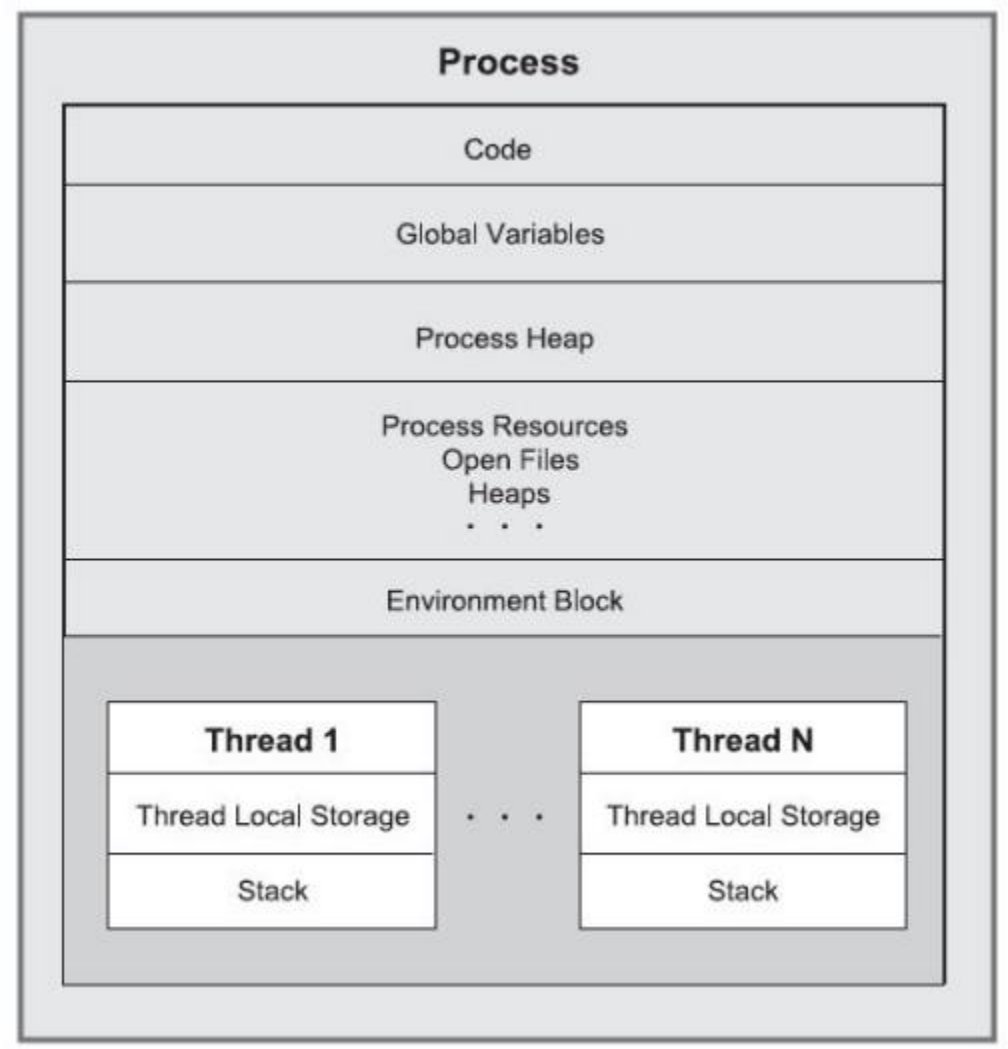
Resources of Processes

From programmer's perspective each process has resources such as one or more threads distinct virtual address space. Although, processes can share memory and files but the process itself lie in an individual virtual memory space.

- One or more code segment including DLLs
- One or more data segment including global variables
- Environment block containing environment variables (current path)

Resources of Threads

Each thread is a unit within the process. Nevertheless, it has resources of its own. Stack: for procedure calls, interrupts, exception handling, and auto variables. Thread Local Storage (TLS): An array like collection of pointers enabling a thread to allocate storage to create its unique data environment. An argument on stack unique for the created thread. A structure containing the context (internal registers status) of the thread.



Lesson 88

Topic 88: **Process Creation**

The most fundamental process management function in windows is CreateProcess(). Windows does not have any structure that keeps account of the parent-child processes. The process that creates a child process is considered as parent process. CreateProcess() has 10 parameters. It does not returns a HANDLE. Rather two handles, one for process and one for thread is returned in a parameter struct.

It creates a process with single primary thread. Windows does not have any structure that keeps account of the parent-child processes. The process that creates a child process is considered as parent process.

CreateProcess() has 10 parameters. It does not return a HANDLE. Rather two handles, one for process and one for thread is returned in a parameter struct. One must be very careful while closing both these handles when they are not needed. Closing the thread handle does not terminate the thread it only deletes the reference to the thread within the process.

BOOL CreateProcess(LPCTSTR lpApplicationName, LPTSTR lpCommandLine,

LPSECURITY_ATTRIBUTES lpProcessAttributes, LPSECURITY_ATTRIBUTES

lpThreadAttributes, BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID

lpEnvironment, LPCTSTR lpCurrentDirectory, LPSTARTUPINFO lpStartupInfo,

LPPROCESS_INFORMATION lpProcessInformation);

lpApplicationName and lpCommandLine specify the program name and the command line parameters. lpProcessAttributes and lpThreadAttributes points to the process and thread's security attribute structure. bInheritHandles specifies whether the new process inherits copies of the calling process's inheritable handles. dwCreationFlags combines several flags

CREATE_SUSPENDED: indicates that the primary thread is a suspended thread and will continue if the process invokes ResumeThread()

DETACHED_PROCESS and CREATE_NEW_CONSOLE are mutual exclusive, First flag creates a process without console and second one creates a process with console

CREATE_UNICODE_ENVIRONMENT: should be set if Unicode is being used.

CREATE_NEW_PROCESS_GROUP : Indicates that the new process is the root of new process group. All processes in the same root group receive the control signal if they share the same console.

lpEnvironment : points to an environment block for the new process. If NULL, the process uses the parent's environment.

lpCurDir Specifies the drive and directory of new process. If NULL parent working directory is used.

lpStartupInfo : specifies the main window appearance and standard device handles for new programs.

lpProcInfo is the pointer to the structure containing handles and id for process and thread.

Lesson 89

Topic 89: Why Processes need IDs and Handles

IDs are unique to processes for their entire lifetime. ID is invalidated when a process is destroyed. Although, it may be reused by other newly created processes. Alternately, a process can have many handles with different security attributes.

Some process management functions require handles while others require IDs. Handles are required for general purpose handle based functions. Just like file handles and handles for other resources the process handles need to be closed when not required.

The process obtains environment and other information from CreateProcess() call. Once the process has been created then changing this information may have no effect on the child. For example the parent process may change its working directory but it will not have effect on child unless the child changes its own working directory. Processes are entirely independent.

Lesson 90

Topic 90: Specifying the Executable Image and the Command Line

Executable Image

Either lpApplicationName or lpCommandLine specifies the executable image. Usually lpCommandLine is used, in which case lpApplicationName is set to NULL. However if lpApplicationName is specified there are rules governing it.

lpApplicationName

lpApplicationName should not be NULL. It should specify the full name of the application including the path. Or use the current path in which case current drive and directory will be used. Include the extension i.e. .EXE or .BAT in the file name. For long names quotes within the string are not required.

lpCommandLine

lpApplicationName should be NULL. Tokens within string are delimited by spaces. First token is the program image name. If the name does not contain the path of the image then the following search sequence is followed.

- Directory of current process image
- The current Directory
- Windows System directory which can be retrieved from GetSystemDirectory()
- Windows Directory which is retrievable from GetWindowsDirectory(). Directories as specified in Environment variable PATH

Command Line

A process can obtain its command line from the usual argv mechanism. Alternately, in windows it can call GetCommandLine(). It's important to know that command line is not a constant string. A program can change the command line. It's advisable that the program makes changes on a copy of command line.

Lesson 91

Topic 91: Inheritable Handles

Mostly, a child process requires access to a resource referenced in parent by a handle. If the handle is inheritable then the child can directly receive a copy of open handles in parent. For example standard input and output handles are shared in this pattern. This inheriting of handles is accomplished in this manner. The bInheritHandles flag in the CreateProcess() call determines whether the child process will inherit copies of parent open handles. Also it is necessary to make individual handles inheritable. Use the SECURITY_ATTRIBUTES structure to specify this. It has a flag bInheritFlag which should be set to TRUE. Also the nLength should be set to sizeof(SECURITY_ATTRIBUTES)

The following code illustrates how this is done programmatically.

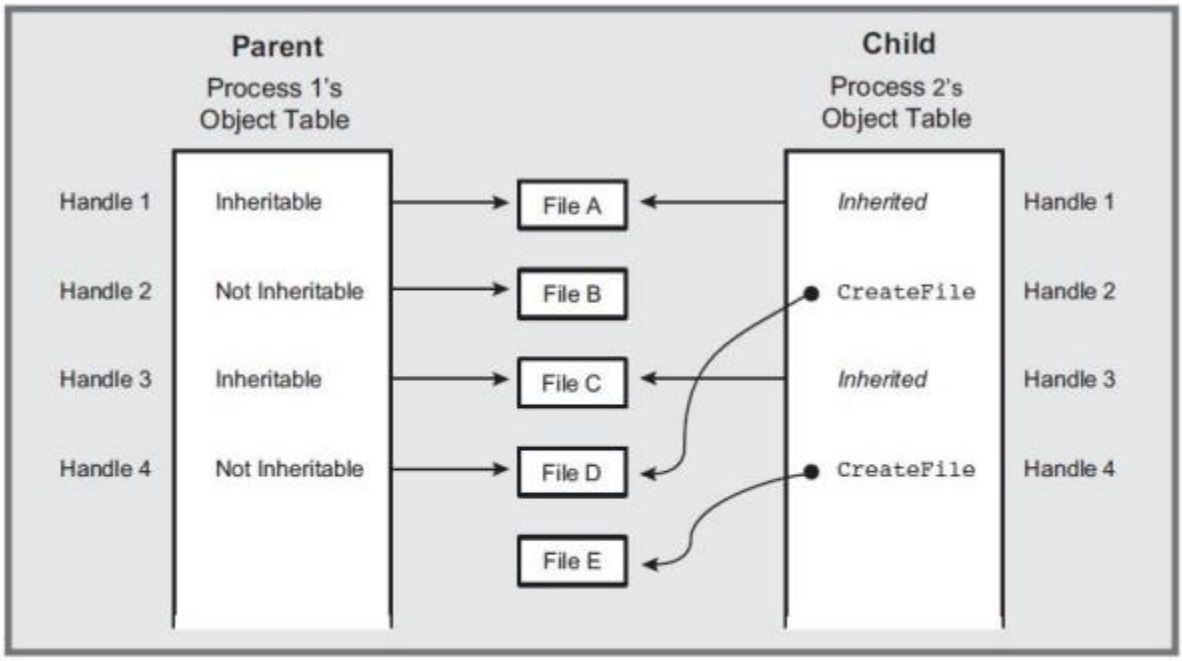
```
HANDLE h1, h2, h3;
SECURITY_ATTRIBUTES sa =
    {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
...
h1 = CreateFile (..., &sa, ... ); /* Inheritable. */
h2 = CreateFile (..., NULL, ... ); /* Not inheritable.
h3 = CreateFile (..., &sa, ...);
    /* Inheritable. You can reuse sa. */
```

Lesson 92

Topic 92: **Passing Inheritable Handles**

We have only made the handles inheritable. However, we need to pass the actual values of handles to the child process. Either this is accomplished through InterProcess Communication (IPC) Or it is passed on the child process by setting it up in the STARTUPINFO struct. The latter is a preferred policy as it allows IO redirection and no changes are required in child process. Another approach is to convert the file handles into text and pass them through the command line to the child process.

The handles if are already inheritable then they will readily accessible to the child. Inherited handles are distinct copies. Parent and child might be accessing same file with different file pointer. Each process should close handles.



- Case of A and C
- Case of B and E
- Case of D
- Also a process itself can have multiple handles to an object.

Lesson 93

Topic 93: **Process Identities**

Process identity can be obtained from the PROCESS_INFORMATION structure. The CreateProcess() API returns the process information in its last parameter. This structure contains the child process handle and id as well as the primary thread handle and id. Closing the child process handle does not destroys the process. It only closes the access of the parent process to the child.

Let's consider these pair of functions for this purpose:

HANDLE GetCurrentProcess (VOID)

DWORD GetCurrentProcessId(VOID)

GetCurrentProcess() is used to create a psuedohandle. It's not inheritable and desired access attributes are undefined. The actual handle can be obtained using Process Id. The OpenProcess() function is used to obtain the process handle using the process ID obtained by GetCurrentProcessId() or otherwise.

HANDLE OpenProcess(DWORD dwDesiredAccess,

BOOL bInheritHandle, DWORD dwProcessId);

dwDesiredAccess provides the desired access attributes for the handle. Some of the commonly required values can be

SYCHRONIZE : Allows process synchronization using wait()

PROCESS_ALL_ACCESS : All access flags are set.

PROCESS_TERMINATE: It's possible to terminate the process with TerminateProcess

PROCESS_QUERY_INFORMATION: The handle can be used to query process information

bInheritHandle is used to specify whether the new process handle can be inherited.

The full pathname of the executable file used to run the process can be determined from

GetModuleFileName()

GetModuleFileNameEx()

VUBWN Contact Us for LMS Handling NASIR ABBAS