Updated Handouts 20/04/2023.

Compiled, updated, organized by Mueen Hasan

MCS 2021-2023

System Programming

CS609

Edition 3rd

Last Updated: 20/04/2023

See the end of this file for changes made in this edition.

Virtual University of Pakistan

Compiled and provided by Mueen Hasan 0301-7923536
Mr.mueenhasan@gmail.com

Table of Contents

Compiled and provided by Mueen Hasan 0301-7923536
Mr.mueenhasan@gmail.com

Compiled and provided by Mueen Hasan 0301-7923536
Mr.mueenhasan@gmail.com

Mueen Hasan 0301-7923536

Week 01: Topic 1-12

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 OS, 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.

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, **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.

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.

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.

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.

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.

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.

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.

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(hO
        ut); 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.

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;
}
```

Topic 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)
- Supportsvery huge file size

File Allocation Tables

- FAT12, FAT16, FAT32 (Start version of File Systems, also supported by Windows)

Compact Disk File System (CDFS)

Supports Universal Disk Format (UDF) & Live File System (LFS) also

Lecture 12: File Naming Conventions

- Windows OS supports several 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.

Week 02: Topic 13 to 24

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:

1. **lpFileName:** It is a string pointer that points to a filename to be opened or created.
2. **dwDesiredAccess:** It is a 32-bit double word which specifies the GENERIC_READ and WRITE access.
3. **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.
4. **lpSecurityAttributes:** points to a security attributes structure.
5. **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.
6. **dwFlagsAndAttributes:** It signifies attributes of the newly created file.
7. **hTemplateFile:** It is the Handle of an open file that specifies attributes to apply to a newly created file.

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

1. **HANDLE** hFile is the file handle.
2. **LPVOID lpBuffer** is the address of the array that stores the data read from the file.
3. **DWORD nNumberOfBytesToRead** is the number of bytes to be read from the file.
4. **LPDWORD lpNumberOfBytesRead** is the number of bytes actually read.
5. **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.

Lecture 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.

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.

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.

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.

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.

Lecture 20: Example for Reporting Errors

The following two header files must be included for different functions, prototypes, and data types.

<environment.h> and <everything.h>

<environment.h> includes a Unicode macro for the environment of the program.

<everything.h> includes all the header files required for typical Windows program.

Example:

```
#include<everything.h>
VOID ReportError(LPCTSTR userMessage, DWORD exitCode, BOOL printErrorMessage)
{
        DWORD eMsgLen, errNum,=GetLastError();
        LPTSTR lpvSysMsg;
    _ftprintf(stderr,                _T("%s\n"),
     userMessage) if (printErrorMessage)
    {
        eMsgLen        =        FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER        |
        FORMAT_MESSAGE_FORM_SYSTEM, NULL, errNum, MAKELANGID(LANG_NEUTRAL,
        SUBLANG_DEFAULT), (LPTSTR) &lpvSysMsg, 0, NULL);
        if (eMsgLen > 0)
        {
                _fprintf(stderr, _T("%s\n"), lpvSysMsg);
        }
        else
        {
                _ftprintf (stderr, _T("Last Error Number; %d\n"), errNum);
        }
       if (lpvSysMsg != NULL) LocalFree(lpvSysMsg);
     }
    if (exitCode>0)
    {
        ExitProcess(exitCode); return;
     }
}
```

Here this function **ReportError()** receives three parameters. Inside this function, the result of **GetLast()** function is stored in the variable, errNum. A generic string variable "lpvSysMsg" is declared for storing the error message.

f the user likes to print the error message, then the error code will first format and convert it into a string using the **FormatMessage()** function.

If message length is greater than 0, then it will print the message, otherwise will print the error code and memory is deallocated.

Topic 20: Reporting Errors

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
*/
#e
lse
#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;
}
```

Topic 21: - Standard IO devices:

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 manipulated 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 succeeds.
- In case of **failure**, it returns **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 **returns true** if calls succeed and return **false in case of fail**.

Topic 22: Copying multiple files using windows API.

We will see in this module how to display files using console 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 takes a variable list of parameters and use to parse these variable lists.

Basically, we specify the list of parameters of a program on command prompt, there are number of options in it. Option () is used to parse these options. It identifies 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 opens these files 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 uses 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;
}
```

Topic 23: Encrypting files:

Encryption is a very old technique, and roman empire use to encrypt secret conversation in war days, and they use Ceasar 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;

}
```

Topic 24: Windows File Management

Windows provides lots of function for file and directory management. These functions are 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.**

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.
- **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

Week 03: Topic 25-36

Topic 25: Windows Directory Management

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's.

```
BOOL SetCurrentDirectory(LPCTSTR lpPathName );
```

```
DWORD GetCurrentDirectory(
                DWORD  nBufferLength,
                LPTSTR lpBuffer );
```

Topic 26: Console I/O

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 );
```

1. **hConsoleHandle** is the console handle which must have GENERIC_WRITE access.
2. **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 );
```

1. **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.

2. **lpBuffer [out]**

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

3. 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.

4. lpNumberOfCharsRead [out]

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

5. lpReserved [in, optional]

This parameter can be NULL.

Topic 27: Printing and Prompting on Console

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;
}
```

Topic 28: 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;

}
```

Topic 29: (64-bit File System)

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.

Topic 30: (File Pointer)

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 );
```

1. **hFile**

   A handle to the file.

2. **lDistanceToMove**

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

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

4. **dwMoveMethod**
   The starting point for the file pointer moves.

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.

Topic 31: (Arithmetic for File)

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);
```

1. **hFile**

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

2. **liDistanceToMove**

The number of bytes to move the file pointer.

3. **lpNewFilePointer**

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

4. **dwMoveMethod**

The starting point for the file pointer move.

- FILE_BEGIN
- FILE_CURRENT
- FILE_END

Topic 32: (Specifying File Pointer Position using Overlapped Structure)

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);
```

Topic 33: (Getting File Size)

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 );
```

1. **hFile**

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

2. **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.

Topic 34: (Random Record Updates using File Pointer)

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 anexample */
}
                    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;
```

Topic 35: (File Attributes and Directory Processing)

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

FindFirstFile()API

```
Handle FindFirstFile(
                    LPCSTRlpFileName,
                    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);
```

Topic 36: (More on File and Directory Attributes)

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);
```

1. **hFile** is a handle to the file or directory.
2. **lpCreationTime** is a pointer to a FILETIME structure to receive the data and time the file or directory was created.
3. **lpLastAccessTime** is a pointer to a FILETIME structure to receive the data and time the file or directory was last accessed.
4. **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 are:

- 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**.

Week 04: Topic 37-47

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);
```

1. **lpPathName** represents the directory path for the filename. The string cannot be longer than 14 characters.
2. **lpPrefixString** represents the null-terminated prefix string. The first 3 characters of this string is used as a prefix of the filename.
3. **uUnique** represents an unsigned integer to be used in creating the temporary filename.
4. **lpTempFileName** is a pointer to the buffer that receives the temporary filename.

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 options 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);
```

Topic 39: Setting File Times

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.

Topic 40: File Processing Strategies

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.

Topic 41: File Locking

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**.

Topic 42: Releasing file locks.

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.

Topic 43: Lock Logic consequences

- 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 process can only ready the file but cannot write into it.

Topic 44: The Registry

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.

Topic 45: Registry Keys

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 sub- keys. 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 pre-loaded by the kernel from a stored "hive", or to the content of a sub-key within another root key, or mapped to a registered service or DLL that provides access to its contained sub-keys and values.

E.g. **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows** refers to the sub-key "Windows" of the sub-key "Microsoft" of the sub-key "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)

Topic 46: Key Management

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>

Topic 47: Listing Registry Keys

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_USER
Get-ChildItem - Path Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
Get- ChildItem -Path Registry::HKCU
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKCU
Get-ChildItem HKCU:
```

Week 05: Topic 48-57

Topic 48: Exception and their Handlers

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.

Topic 49: Try and Except Blocks

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.

Topic 50: Filter Expressions

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

Topic 51: Exceptions Codes

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.

Topic 52: Exceptions Handling Sequence

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.

Topic 53: Floating-Point Exceptions

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

Topic 54: Errors and Exceptions

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.

Topic 55: Treating Errors as Exceptions

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

- A fatal error terminates the entire process when only a single thread should terminate.
- The programmer may require continuing program execution rather than terminate the process.
- Synchronization resources, such as events or semaphores, will not be released in most of the circumstances.

Topic 56: Termination Handlers

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.

Topic 57: Better Programs with Termination Handlers

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));
```

Week 06: Topic 58-69

Topic 58: Filter Functions

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.

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 final 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.

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

Topic 59: Console Control Handles

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.

Topic 60: Console Control Handler Example

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.

Topic 61: Vectored Exceptions Handling

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 look 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.

Topic 62: Memory Management

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.

Topic 63: Windows Memory Management Overview

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.

Topic 64: Introduction to Heaps

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.

Topic 65: Creating Heaps

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.

Topic 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);
```

1. **hHeap** is the handle of the heap from which memory is to be allocated. dwFlags are quite similar to the flags used in HeapCreate().
2. **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.
3. **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.

4. **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);
```

1. **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().
2. **HEAP_ZERO_MEMORY:** only the newly allocated memory is set to zero (in case dwBytes is greater than the previous allocation).
3. **HEAP_REALLOC_IN_PLACE_ONLY:** It indicates the newly allocated block should lie contiguously to the previously allocated block.
4. **lpMem:** specifies the pointer to the block previously allocated to the same heap hHeap.
5. **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.

Topic 67: Heap Size and Serialization

Some programs may require determining 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.

Topic 68: Using Heaps

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.

Topic 69: Working with Heaps

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.

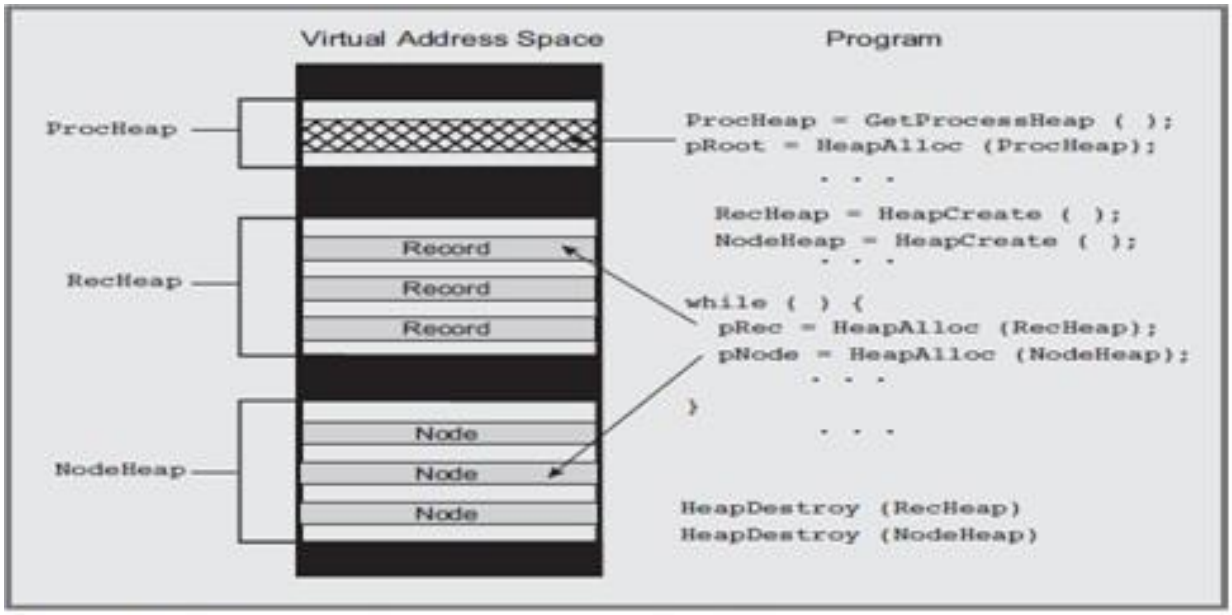


Figure 5–2 Memory Management in Multiple Heaps

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().

Week 07: Topic 70-81

Topic 70: Binary Search Using Heaps

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 of 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. */
```

Program code is skipped due to very large program as such program are not important from exam point of view.

Topic 71: Memory-Mapped Files

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.

Topic 72: File Mapping Objects

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 );
```

Note: API is **CreateFileMapping** and return type is **Handle** and number of arguments are **6**.

- ✓ **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.

Topic 73: Open Existing File Mapping Objects

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,
                        LPCTSTR lpMapName );
```

Note: API is **OpenFileMapping** and return type is **Handle** and number of arguments are **3**.

- ✓ **dwDesiredAccess** is checked against the accessed assigned by **CreateFileMapping**();
- ✓ **lpMapName** is the name of the mapping assigned by **CreateFileMapping**().

Topic 74: Mapping Objects to Process Address Space

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  );
```

Note: API is **MapViewOfFile** and number of arguments are **5**.

Return: The starting address of the block (file view) on success, or **NULL on failure**.

1. **hFileMappingObject** is file mapping object previously obtained by **CreateFileMapping**().
2. **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
3. **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.
4. **dwNumberOfBytesToMap** shows the number of bytes of file to map. Set as zero to map the whole file.

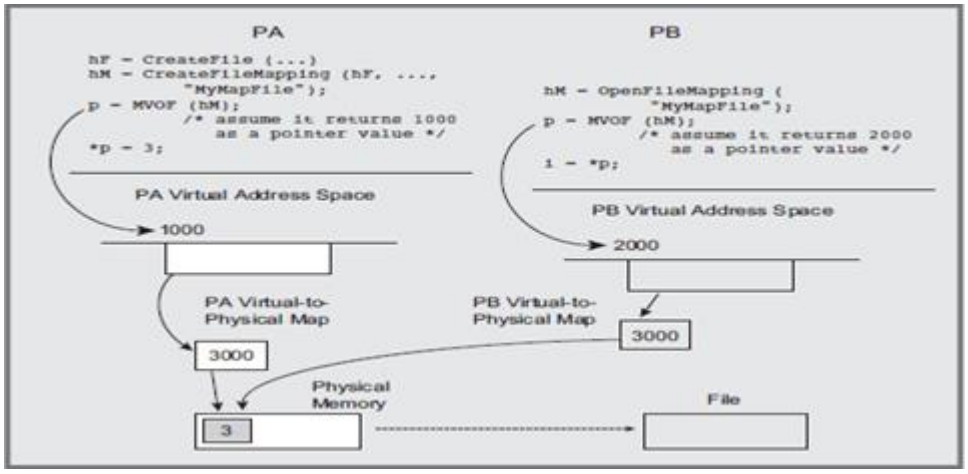
If the function is successful, it returns the base address of the mapped region. If the function fails, the return value is NULL.

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()**.

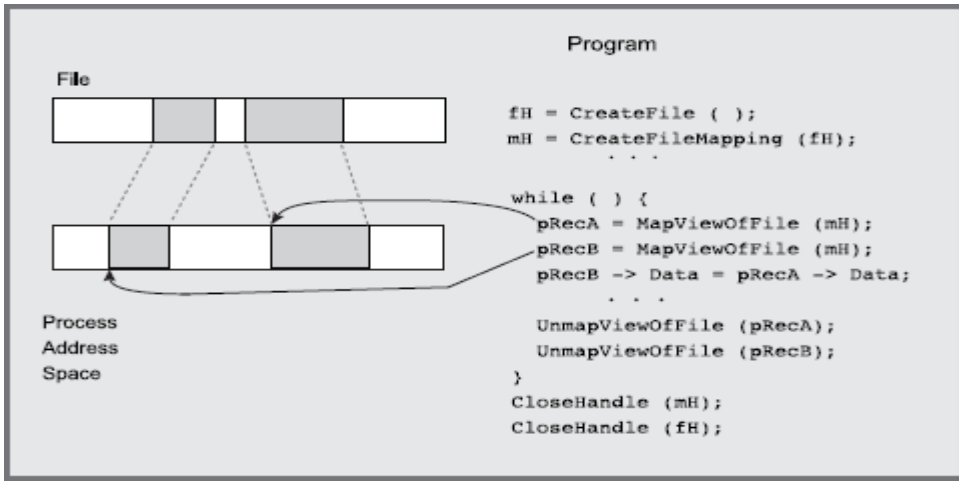
```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress );
```

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.



Topic 75: More About File Making

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.

Topic 76: Sequential File Access Using file mapping

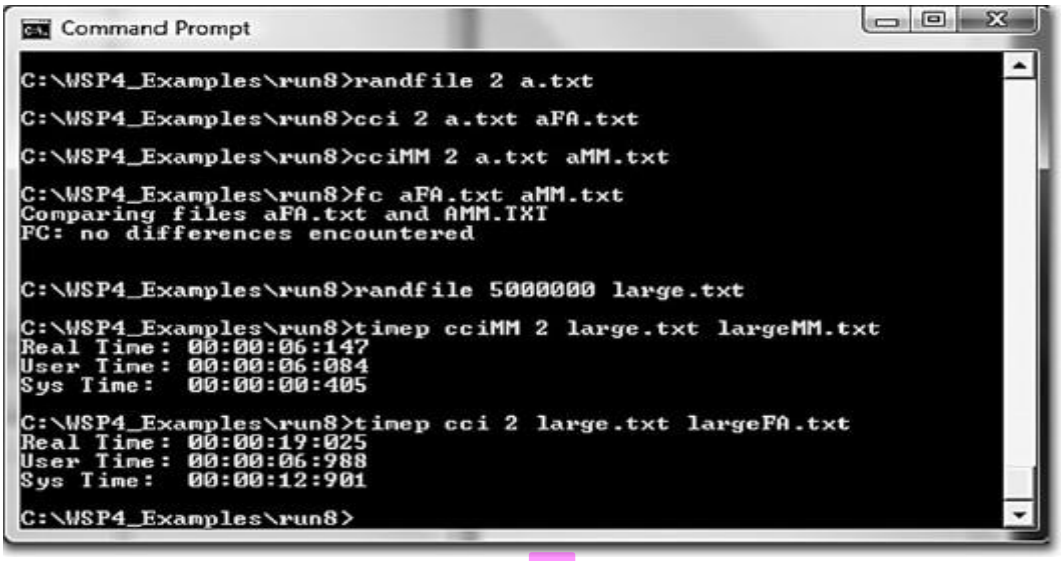
Sequential File Access

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.

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.



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.

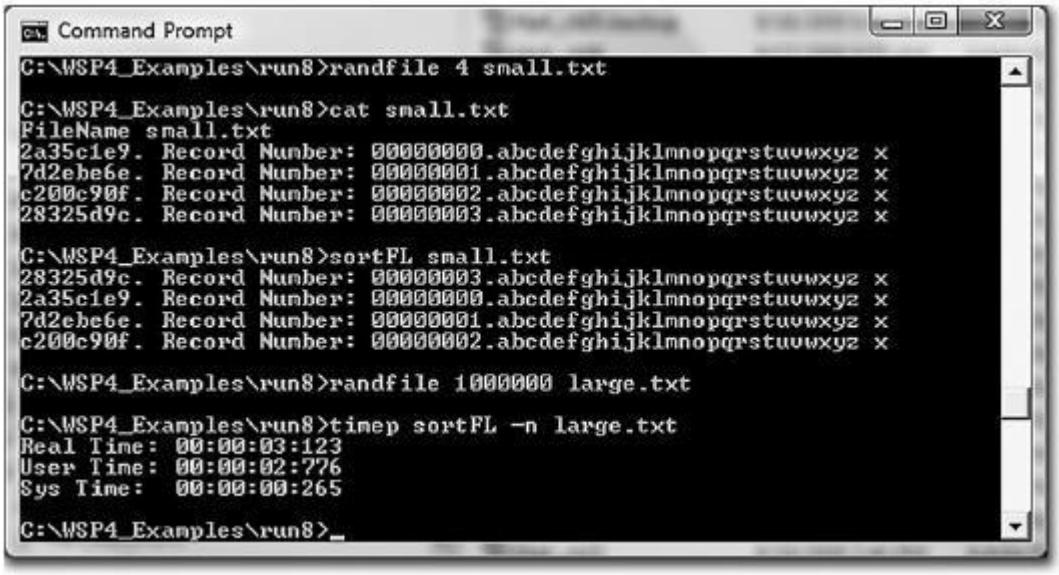
Program: Sorting a File with Memory Mapping #include<everything.h>

typedef struct _RECORD

{

```
        TCHAR    key[KEY_SIZE];
        TCHAR data[DATALEN]
} RECORD;


fdefine REx:SIZE  sizeof  (REX»IID)


J.nt  _tmain  (J.nt  «rgc,  LPl'S'l'R  <>rgv[])
{
    HANDLE  hFile = DlVALID_HAIIDLE_V.AIJUE,  hMa.p =  !TULL;
    LPVOID  pFile = NULL;
    Ll\RGE_INTEGER
    fJ.l<lS!.ze;            '!'CHAR
    tempFile[       lAX_PATIl);
    LPTSTR pTFile;

    /*croo.te the  name for   a  temporary  file  to hold
        a  oopy of the  fJ.le t,obe  so-rted.  SortJ.ng 1.s
        done in  the  temp fJ.le.
    _stprintf   (t.empFile,  _T  ("'!>s.t.mp"),
    «rgv[ll);          CopyFile          (a.rgv[1],
                          teIJ\pFile,  TRUE);
    /*  M«p the  temporary   fJ.le «ndsort  it   J.nmemory.  */
    hFile  =croo.teFile  (tempFile,   GEllERIC_READ   I GEllERIC_NRITE,
            0,  !TULL, OPE!l_EXISTING, 0,  llUL[.);
    GetFileSiZeElC              (hFile,
    &filesize);  fileS!.ze.  dPart
    += 2;
     lap =croo.t.eFil lappJ.ng (hFile, !TULL, PAGE_REAIX'ffiITE,
            fil<lSize.HighP<\rt,  fil<lSize.LOWP<>rt,  lfULL);
    pFile = MapViewOfFile (hMap, FILE_MAP_ALL_.AO::ESS, O,  O,  0);
```

```
    qsort (pFile, FsLow / REx:SIZE1 REx:SIZEr KeyCo!J\pare);
                          f* Keycompa.re is «sin PrOgram :;-2.

    /* Print the sorted file. */
    pTFile = (LPTSTR) pFile;
    pTFile[fileSize.QUadPart/TSIZEJ =
    '\O';
    _tprJ.ntf (_T ("'!>S")'
    pFJ.le);
    unmapviewo:fFile
    (pFile); CloseHandle
    (hllap); closeHandle
    (hFile); DeleteFJ.le (
    tempFile); return O;
}
```



Topic 78: Using Base

Pointer 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.

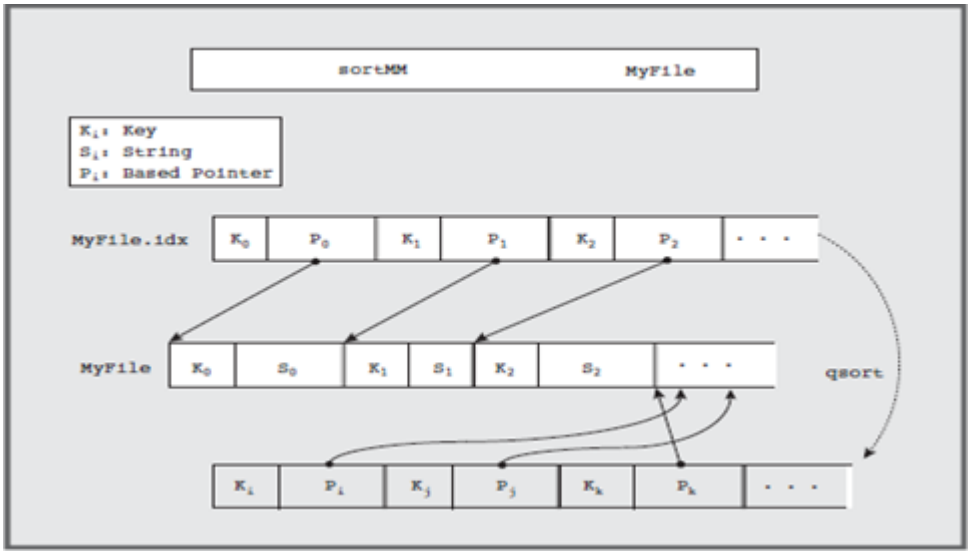
Topic 79: Base Pointer Example

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.



Topic 80: Dynamic Link Library

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 reduce 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.

Topic 81: Implicit Linking

Implicit linking is the easier of the two techniques. The process is as follows.

1. Functions defined in a DLL are collected and build as DLL.
2. 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.
3. 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.
4. Function interfaces defined in DLLs should be exported carefully.

Week 08: Topic 82-92

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.

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 );
```

1. **hModule** is an instance produced by **LoadLibrary**.
2. **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.

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. */
```

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)
```

1. **hDll** corresponds to the handle returned by **LoadLibrary().**
2. **Reason** will have on the four values.
   - DLL_PROCESS_ATTACH
   - DLL_THREAD_ATTACH
   - DLL_THREAD_DETACH
   - DLL_PROCESS_DETACH
- **lpReserved** if NULL, represent explicit attachment else it represents implicit attachment.

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.

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 requires 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).

Topic 87: Windows Processes and Threads

Windows Process Management is all about:

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

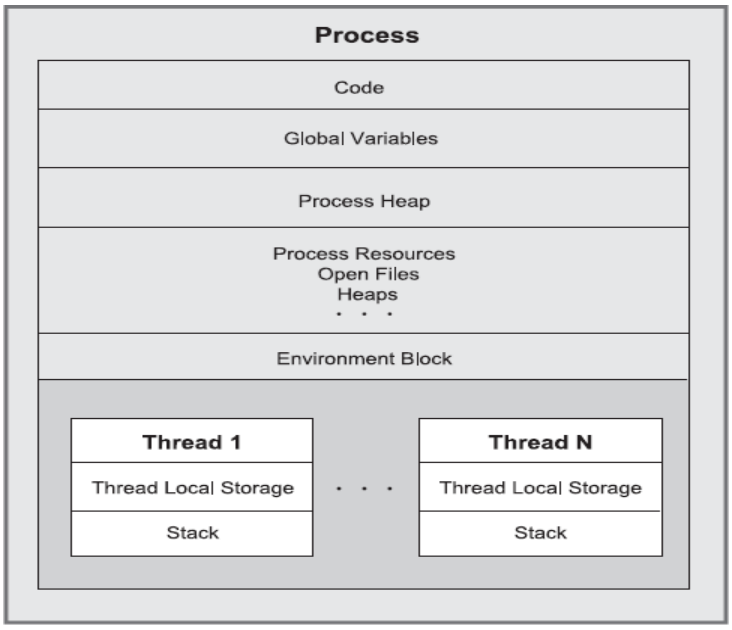
Resources of Processes

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.



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 return 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 );
```

1. lpApplicationName and lpCommandLine specify the program name and the command

line parameters.

2. **lpProcessAttributes** and lpThreadAttributes points to the process and thread's security attribute structure.
3. **bInheritHandles** specifies whether the new process inherits copies of the calling process's inheritable handles.
4. **dwCreationFlags** combines several flags which are given below.
    - **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 is mutual exclusive, First flag creates a process without console and second one creates a process with consol.
    - **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.
5. **lpEnvironment** : points to an environment block for the new process. If NULL, the process uses the parent's environment.
6. **lpCurDir** Specifies the drive and directory of new process. If NULL parent working directory is used.
7. **lpStartupInfo** : specifies the main window appearance and standard device handles for new programs.
8. **lpProcInfo** is the pointer to the structure containing handles and id for process and thread.

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.

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.

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)

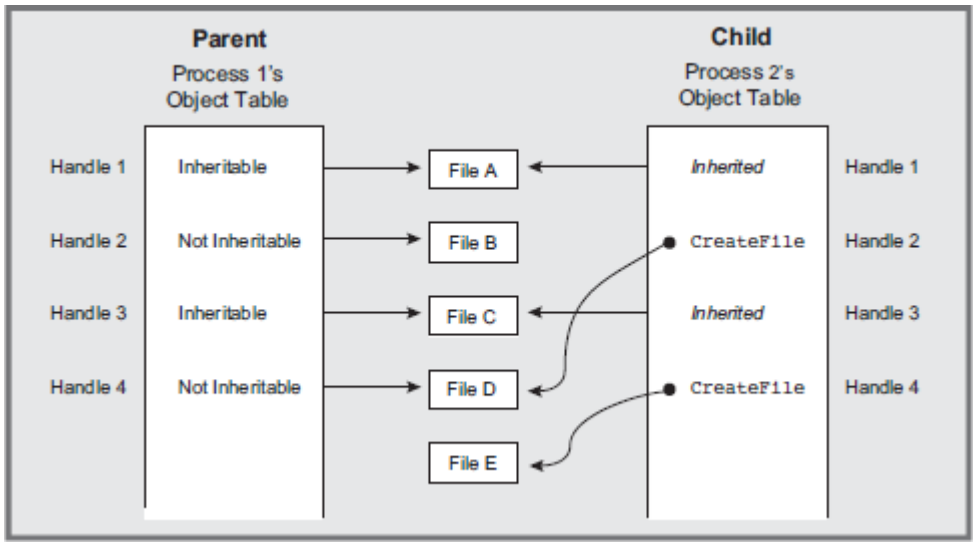
The following code illustrates how this is done programmatically.

```
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. */
```

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 be 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.



Week 09: Topic 93-105

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 destroy 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()**

Topic 94: Duplicating Handles

Parent and Child processes may require different access to an inheritable handle of resources. Parent Process may also require real handle rather than a **pseudohandle**. Parent process usually creates a duplicate handle to take care of this problem. Duplicate handles are created using API **DuplicateHandle().**

```
BOOL DuplicateHandle (
                    HANDLE hSourceProcessHandle,
                    HANDLE hSourceHandle,
                    HANDLE hTargetProcessHandle,
                    LPHANDLE lpTargetHandle,
                    DWORD dwDesiredAccess,
                    BOOL bInheritHandle,
                    DWORD dwOptions  );
```

1. **hSourceProcessHandle** is the source process handle.
2. **lpTargetHandle** receives a pointer to a copy of the original handle indicated by hSourceHandle. The parent process duplicating the handle must have PROCESS_DUP_HANDLE access. **If the hSourceHandle does not exist in the source process, then the function will fail**. The target handle received will only be valid in the process whose handle is specified in hTargetProcessHandle.

   Usually, the handle is duplicated by the source or target process. In case it is duplicated by some other process than a mechanism is required to communicate with the target process DuplicateHandle() can be used to duplicate any sort of handle.
3. **dwOption** can be combination of two flags.
   - **DUPLICATE_CLOSE_SOURCE:** causes the source handle to be closed.
   - **DUPLICATE_SAME_ACCESS:** Uses the access rights of the source handle.
4. **dwDesiredAccess** is ignored, Reference Count. Windows keep track of all the handles created for a resource. This count is not accessible to the application. A resource cannot be released unless the last reference to the resource has been closed. Inherited and duplicated handles reflect on reference count.

Topic 95: Exiting and Terminating a Process

A process can exit using the API **ExitProcess()**. The function does not return, in fact the process and all its associated threads terminate. Termination Handlers are ignored. However, a detach call is sent to DllMain().

```
VOID ExitProcess(UINT uExitCode) ;
```

Exit Code

The exit code of the process can be determined using the API GetExitCodeProcess().

```
BOOL GetExitCodeProcess(
                        HANDLE hProcess,
                        LPDWORD lpExitCode );
```

The process identified by hProcess must have PROCESS_QUERY_INFORMATION flag set. lpExitCode points to the dword that will receive the exit code of the given process.

One possible value of Exit code can be STILL_ACTIVE

Terminate Process

One process can terminate another process using the function **TerminateProcess()**.

```
BOOL TerminateProcess (
                        HANDLE hProcess,
                        UINT uExitCode );
```

hProcess is the handle to the process to be terminated uExitCode is the code returned by the process on termination.

Exiting and Terminating Process

Process must be exited and terminated carefully. All the handles need to be closed before exiting the process. It's not a good idea to use **ExitProcess()** within the program as it will not give it chance to release resources. Use is with Exception handling. TerminateProcess() does not all the process to execute SHE or DllMain().

Topic No.96: (Waiting for a Process)

The synchronization can be attained easily by the wait process. Windows provides a general- purpose wait function. Which can be used to wait for a single object and also multiple objects in a group. Windows send a signal to the waiting process when the process terminates.

We have two API's

```
DWORD waitForSingleObject(
                                    HANDLE hHandle,
                                    DWORD dwMilliseconds);
```

```
DWORD waitForMultipleObjects(
                                    DWORD nCount,
                                    CONST HANDLE*lpHandles,
                                    BOOL bWaitAll,
                                    DWORD dwMilliseconds);
```

1) **hHandle** is the single object for which the process waits.
2) **lpHandle** is the array of handles for which the process waits.
3) **nCount** is the number of objects in an array. Should not exceed MAXIMUM WAIT OBJECTS
4) **dwMilliseconds** is the timeout period for wait. 0 for no wait and INFINITE for indefinite wait.
5) **bWaitAll** describes if it's necessary to wait for all the objects to get free. The possible return values are:
    - WAIT_OBJECT_0
    - WAIT_OBJECT_0+n
    - WAIT_TIMEDOUT
    - WAIT_FAILED
    - WAIT_ABANDONED_0

Topic No.97: (Environment Block)

- Environment Block string.
- An Environment Block is associated with each process.
- The EB contains string regarding the environment of the process of the form Name = Value
- Each string is NULL terminated.
- PATH is an example of an environment string.
- Environment variables can get and set using these API's

```
DWORD GetEnvironmentVatiable (
                                    LPCTSTR lpNAME,
                                    LPTSTR lpBuffer,
                                    DWORD nSize );
```

```
BOOL SetEnviromentVariable (
                                LPCTSTR lpName,
                                LPCTSTR lpValue);
```

To share the parent environment with the child process set lpEnvironment to NULL in the call to **CreateProcess()**. Any process can modify the environment variables and make new ones.

lpName is the variable name. On setting the string the value is modified if the variable already exists. If it does not exist, then a new variable is created and assigned the value. IF the value is NULL then the variable is deleted.

"=" is reserved and cannot be used in the variable name.

GetEnvironmentVariable() return the length of variable string or 0 in case of failure.

If LpValue is not as long as the value specified by the count, then the actual length of the string is returned.

The access rights given in Create Process () are usually Process_All_Access. But there are several **options**:

- PROCESS_QUERY_INFORMATION
- CREATE_PROCESS
- PROCESS_TERMINATE
- PROCESS_SET_INFORMATION
- DUPLICATE_HANDLE
- CREATE_HANDLE

Topic No.98: (A Pattern Searching Example)

A pattern searching example: This example uses the power of windows multitasking to search a specific pattern among numerous files.

The process takes the specific pattern along with filenames through command line. The standard output file is specified as inheritable in new process start-up info structure. Wait functions are used for synchronization. As soon as the search end the results are displayed one at a time. Wait functions are limited for 64 handles so program works accordingly. The program uses the exit code to identify whether the process has detected a match or not.

The Example

```
#include "Everything.h"
int _tmain (int argc, LPTSTR argv[])
/*      Create a separate process to search each file on the command line. Each process is given a temporary file, in the current directory, to receive the results. */
{
HANDLE hTempFile;
SECURITY_ATTRIBUTES stdOutSA = /* SA for inheritable handle. */
{sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
TCHAR commandLine[MAX_PATH + 100]; STARTUPINFO startUpSearch, startUp; PROCESS_INFORMATION processInfo; DWORD exitCode, dwCreationFlags = 0; int iProc;
HANDLE *hProc; /* Pointer to an array of proc handles. */ typedef struct {TCHAR tempFile[MAX_PATH];} PROCFILE; PROCFILE *procFile; /* Pointer to array of temp file names. */
#ifdef UNICODE
dwCreationFlags = CREATE_UNICODE_ENVIRONMENT; #endif
if (argc < 3)
ReportError (_T ("Usage: grepMP pattern files."), 1, FALSE);
/* Startup info for each child search process as well as the child process that will display the results. */
GetStartupInfo (&startUpSearch); GetStartupInfo (&startUp);
/* Allocate storage for an array of process data structures,
each containing a process handle and a temporary file name. */ procFile = malloc ((argc - 2) * sizeof (PROCFILE));
```

hProc = malloc ((argc - 2) * sizeof (HANDLE));

/* Create a separate "grep" process for each file on the command line. Each process also gets a temporary file

name for the results; the handle is communicated through the STARTUPINFO structure. argv[1] is the search pattern. */

for (iProc = 0; iProc < argc - 2; iProc++) {

/* Create a command line of the form: grep argv[1] argv[iProc + 2] */

/* Allow spaces in the file names. */

_stprintf (commandLine, _T ("grep \"%s\" \"%s\""), argv[1], argv[iProc + 2]);

/* Create the temp file name for std output. */

if (GetTempFileName (_T ("."), _T ("gtm"), 0, procFile[iProc].tempFile) == 0) ReportError (_T ("Temp file failure."), 2, TRUE);

/* Set the std output for the search process. */ hTempFile = /* This handle is inheritable */

CreateFile (procFile[iProc].tempFile,

/** GENERIC_READ | Read access not required **/ GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, &stdOutSA, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

if (hTempFile == INVALID_HANDLE_VALUE)

ReportError (_T ("Failure opening temp file."), 3, TRUE);

/* Specify that the new process takes its std output from the temporary file's handles. */

startUpSearch.dwFlags = STARTF_USESTDHANDLES;

startUpSearch.hStdOutput = hTempFile; startUpSearch.hStdError = hTempFile;

statUpSearch.hStdInput = GetStdHandle (STD_INPUT_HANDLE);

/* Create a process to execute the command line. */ if (!CreateProcess (NULL, commandLine, NULL, NULL,

TRUE, dwCreationFlags, NULL, NULL, &startUpSearch, &processInfo)) ReportError (_T ("ProcCreate failed."), 4, TRUE);

/* Close unwanted handles */

CloseHandle (hTempFile); CloseHandle (processInfo.hThread);

```
/* Save the process handle. */
hProc[iProc] = processInfo.hProcess;
}
/* Processes are all running. Wait for them to complete, then output
the results - in the order of the command line file names. */ for (iProc = 0; iProc
< argc-2; iProc += MAXIMUM_WAIT_OBJECTS)
WaitForMultipleObjects (min(MAXIMUM_WAIT_OBJECTS, argc - 2 - iProc),
&hProc[iProc], TRUE, INFINITE);
/* Result files sent to std output using "cat".
Delete each temporary file upon completion. */ for (iProc = 0; iProc < argc - 2;
iProc++) {
if (GetExitCodeProcess (hProc[iProc], &exitCode) && exitCode == 0) {
/* Pattern was detected - List results. */
/* List the file name if there is more than one file to search */ if (argc > 3) _tprintf
(_T("%s:\n"), argv[iProc+2]);
_stprintf (commandLine, _T ("cat \"%s\""), procFile[iProc].tempFile); if
(!CreateProcess (NULL, commandLine, NULL, NULL,
TRUE, dwCreationFlags, NULL, NULL, &startUp,
&processInfo))
ReportError (_T ("Failure executing cat."), 0, TRUE);
else {
WaitForSingleObject (processInfo.hProcess, INFINITE); CloseHandle
(processInfo.hProcess);
CloseHandle (processInfo.hThread);
}
}
CloseHandle (hProc[iProc]);
if (!DeleteFile (procFile[iProc].tempFile))
ReportError (_T ("Cannot delete temp file."), 6, TRUE);
}
free (procFile); free (hProc); return 0;
}
```

Topic No.99: Working in Multiprocessor Environment

Multiprocessor Environment

All the processes run concurrently in windows environment. If the system is uniprocessor the processor time is multiplexed among multiple processes in an interleaved manner. If the system is multiprocessor, then windows scheduler can run process threads on separate processors.

There would be substantial performance gain in this case. The performance gain will not be linear because of dependencies among processes (wait and signal).

From programming point of view, it is essential to understand this potential of windows so that programs can be designed optimally. Alternately, constraining the processing to a specific processor is also possible. This is accomplished by processor affinity mask.

Subsequently, it is possible to create independent threads within a process which can be scheduled on separate processor.

Topic No.100: (Working in Multiprocessor Environment)

Process Times

Windows API provides a very simple mechanism for determining the amount of time a process has consumed.

The function used for this purpose is **GetProcessTime()**.

```
BOOL GetProcessTimes (
        HANDLE hProcess,
        LPFILETIME lpCreationTime,
        LPFILETIME lpExitTime,
        LPFILETIME lpKernelTime,
        LPFILETIME lpUserTime );
```

- ✓ Process handle can be of the current process or a terminated one.
- ✓ Elapsed time can be computed by subtracting creation time from exit time.
- ✓ **FILETIME is a 64-bit value.**
- ✓ **GetThreadTime()** can be used similarly and requires a thread handle.

Topic No.101: (Process Execution Times)

Example of Process Execution Times

This example prints the elapsed, real, and user times. It uses the API GetCommandLine()to get the command line as a single string. It then uses the SkipArg() function the skip past the executable name.

```
#include "Everything.h"
int _tmain (int argc, LPTSTR argv[])
{
STARTUPINFO startUp; PROCESS_INFORMATION procInfo;
union {/* Structure required for file time arithmetic. */ LONGLONG li;
FILETIME ft;
} createTime, exitTime, elapsedTime;
FILETIME kernelTime, userTime;

SYSTEMTIME elTiSys, keTiSys, usTiSys; LPTSTR targv, cLine = GetCommandLine ();
OSVERSIONINFO windowsVersion; HANDLE hProc;
targv = SkipArg(cLine, 1, argc, argv);
/* Skip past the first blank-space delimited token on the command line */ if (argc <= 1 || NULL == targv)
ReportError (_T("Usage: timep command ..."), 1, FALSE);
/* Determine is this is Windows 2000 or NT. */ windowsVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
if (!GetVersionEx (&windowsVersion))
ReportError (_T("Can not get OS Version info. %d"), 2, TRUE);
if (windowsVersion.dwPlatformId != VER_PLATFORM_WIN32_NT)
ReportError (_T("This program only runs on Windows NT kernels"), 2, FALSE);
/* Determine is this is Windows 2000 or NT. */ windowsVersion.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
if (!GetVersionEx (&windowsVersion))

ReportError (_T("Can not get OS Version info. %d"), 2, TRUE);
if (windowsVersion.dwPlatformId != VER_PLATFORM_WIN32_NT)
ReportError (_T("This program only runs on Windows NT kernels"), 2, FALSE);
```

```
GetStartupInfo (&startUp);
        /* Execute the command line and wait for the process to complete. */ if
(!CreateProcess (NULL, targv, NULL, NULL, TRUE,
        NORMAL_PRIORITY_CLASS, NULL, NULL, &startUp, &procInfo)) ReportError (_T
("\nError starting process. %d"), 3, TRUE);
        hProc = procInfo.hProcess;
        if (WaitForSingleObject (hProc, INFINITE) != WAIT_OBJECT_0)
        ReportError (_T("Failed waiting for process termination. %d"), 5, TRUE);;

        if (!GetProcessTimes (hProc, &createTime.ft, &exitTime.ft, &kernelTime,
&userTime))
        ReportError (_T("Can not get process times. %d"), 6, TRUE);
        elapsedTime.li = exitTime.li - createTime.li; FileTimeToSystemTime
(&elapsedTime.ft, &elTiSys); FileTimeToSystemTime (&kernelTime, &keTiSys);
FileTimeToSystemTime (&userTime, &usTiSys);
        _tprintf (_T ("Real Time: %02d:%02d:%02d.%03d\n"),
        elTiSys.wHour, elTiSys.wMinute, elTiSys.wSecond,
        elTiSys.wMilliseconds);
        _tprintf (_T ("User Time: %02d:%02d:%02d.%03d\n"), usTiSys.wHour,
usTiSys.wMinute, usTiSys.wSecond, usTiSys.wMilliseconds);
        _tprintf (_T ("Sys Time: %02d:%02d:%02d.%03d\n"), keTiSys.wHour,
keTiSys.wMinute, keTiSys.wSecond,
        keTiSys.wMilliseconds);
        CloseHandle (procInfo.hThread); CloseHandle (procInfo.hProcess);
        return 0;
        }
```

Topic No.102: (Generating Console events)

Example of Process Execution Times

This example prints the elapsed, real and user times. It uses the API **GetCommandLine()**to get the command line as a single string. It than uses the SkipArg() function the skip past the executable name.

```
#include "Everything.h"
int _tmain (int argc, LPTSTR argv[])

{
        STARTUPINFO

        startUp;

        PROCESS_INF

        ORMATION

        procInfo;

        union {         /* Structure required for

                file time arithmetic. */

                LONGLONG li;

                FILETIME ft;

        } createTime, exitTime, elapsedTime;

FILETIME kernelTime, userTime;

        SYSTEMTIME elTiSys,

        keTiSys, usTiSys;

        LPTSTR targv, cLine =

        GetCommandLine ();

        OSVERSIONI

        NFO

        windowsVers
```

```
ion; HANDLE

hProc;

targv = SkipArg(cLine, 1, argc, argv);

/*  Skip past the first blank-space delimited token

on the command line */ if (argc <= 1 || NULL ==

targv)

        ReportError (_T("Usage: timep command ..."), 1, FALSE);

/* Determine is this is Windows 2000 or NT.  */

windowsVersion.dwOSVersionInfoSize =

sizeof(OSVERSIONINFO); if (!GetVersionEx

(&windowsVersion))

        ReportError (_T("Can not get OS Version info. %d"), 2, TRUE);

if (windowsVersion.dwPlatformId != VER_PLATFORM_WIN32_NT)

        ReportError (_T("This program only runs on Windows NT kernels"), 2, FALSE);

/* Determine is this is Windows 2000 or NT.  */

windowsVersion.dwOSVersionInfoSize =

sizeof(OSVERSIONINFO); if (!GetVersionEx

(&windowsVersion))

        ReportError (_T("Can not get OS Version info. %d"), 2, TRUE);

if (windowsVersion.dwPlatformId != VER_PLATFORM_WIN32_NT)

        ReportError (_T("This program only runs on Windows NT

kernels"), 2, FALSE); GetStartupInfo (&startUp);

/* Execute the command line and wait for the

process to complete. */ if (!CreateProcess
```

```
(NULL, targv, NULL, NULL, TRUE,

        NORMAL_PRIORITY_CLASS, NULL, NULL,

        &startUp, &procInfo)) ReportError (_T

        ("\nError starting process. %d"), 3, TRUE);

hProc = procInfo.hProcess;

if (WaitForSingleObject (hProc, INFINITE) != WAIT_OBJECT_0)


ReportError (_T("Failed waiting for process
    termination. %d"), 5, TRUE);;


if (!GetProcessTimes (hProc,

        &createTime.ft,

        &exitTime.ft,

        &kernelTime,

        &userTime))

                ReportError (_T("Can not get process times. %d"), 6, TRUE);


elapsedTime.li = exitTime.li -

createTime.li;

FileTimeToSystemTime

(&elapsedTime.ft, &elTiSys);

FileTimeToSystemTime

(&kernelTime, &keTiSys);

FileTimeToSystemTime
```

```
        (&userTime, &usTiSys);

            _tprintf (_T ("Real Time:
          %02d:%02d:%02d.%03d\n"),


elTiSys.wHour, elTiSys.wMinute, elTiSys.wSecond,

                elTiSys.wMilliseconds);

        _tprintf (_T ("User Time:

                %02d:%02d:%02d.%03d\n"),

                usTiSys.wHour,

                usTiSys.wMinute,

                usTiSys.wSecond,

                usTiSys.wMilliseconds);

        _tprintf (_T ("Sys Time:

                %02d:%02d:%02d.%03d\n"),

                keTiSys.wHour, keTiSys.wMinute,

                keTiSys.wSecond,

                keTiSys.wMilliseconds);


        CloseHandle (procInfo.hThread); CloseHandle

        (procInfo.hProcess); return 0;

}
```

Topic No.103: (Simple job Management Shell)

Terminating another process can be problematic as the terminating process does not get a chance to release resources. SEH does not help either as there is no mechanism that can be used to raise an exception in another process.

Console control event allows sending a message from one process to another. Usually, a handler is set up in a process to catch such signals. Subsequently, the handler generates an exception. It is hence possible for a process to generate an event in another process.

CreateProcess() can be used with the flag CREATE_NEW_PROCESS_GROUP.

This creates a process group in which the created process is the root and all the subsequently created processes by the parent are in the new group.

One process in a group can generate a CTRL_C_EVENT or CTRL_BREAK_EVENT in a group identified by the root process ID.

The target process should have the same console as the process generating the event. More specifically, the calling process cannot have its own console using CREATE_NEW_CONSOLE or DETACHED_PROCESS flags.

```
BOOL GenerateConsoleCtrlEvent(
                                    DWORD dwCtrlEvent,
                                    DWORD dwProcessGroupId );
```

1. **dwCtrlEvent** can be CTRL_C_EVENT or CTRL_BREAK_EVENT
2. **dwProcessGroupId** is the id of the process group.

Topic No.104: (Get a job Number)

Simple Job Management Shell

This job shell will allow three commands to run.

- jobbg
- jobs
- kill

The job shell parses the command line and then calls the respective function for the given command. **The shell uses a user-specific file keeping track of process ID and other related information.**

Several shells can run concurrently and use this shared file.

Also, concurrency issues can be encountered as several shells can try to use the same file. File locking is used to make the file access mutually exclusive.

/* JobObjectShell.c One

program combining three

job management

commands:

Jobbg - Run a job in the background

jobs - List all background jobs.

kill - Terminate a specified job of the job family.

There is an option to generate a console control signal.

This implementation enhances JobShell with a time limit on each process.

There is a time limit on each process, in

seconds, in argv[1] (if present)

0 or omitted means no process

```
                    time limit

*/

#include "Everything.h"
#include "JobManagement.h"
#define MILLION 1000000
static int Jobbg (int,

LPTSTR *, LPTSTR);

static int Jobs  (int,

LPTSTR *, LPTSTR);

static int Kill  (int,

LPTSTR *, LPTSTR);

HANDLE hJobObject

= NULL;

JOBOBJECT_BASIC_LIMIT_INFORMATION basicLimits = {0, 0,
JOB_OBJECT_LIMIT_PROCESS_TIME};
int _tmain (int argc, LPTSTR argv[])

{

LARGE_INTEGER processTimeLimit;

        BOOL exitFlag = FALSE;

        TCHAR command[MAX_COMMAND_LINE], *pc;

        DWORD i, localArgc;

        TCHAR

        argstr[MAX_ARG][MAX_COMMAND_LINE];

        LPTSTR pArgs[MAX_ARG];


        /* NT Only - due to file locking */

        if (!WindowsVersionOK (3, 1))
```

```
ReportError (_T("This program requires Windows NT 3.1 or

greater"), 1, FALSE); hJobObject = NULL;

processTimeLimit.QuadPart = 0;

if (argc >= 2) processTimeLimit.QuadPart = _ttoi(argv[1]);
basicLimits.PerProcessUserTimeLimit.QuadPart = processTimeLimit.QuadPart *
MILLION;

hJobObject = CreateJobObject(NULL, NULL); if (NULL == hJobObject)
ReportError(_T("Error creating job object."), 1, TRUE);

if (!SetInformationJobObject(hJobObject, JobObjectBasicLimitInformation,
&basicLimits, sizeof(JOBOBJECT_BASIC_LIMIT_INFORMATION)))

ReportError(_T("Error setting job object information."), 2, TRUE);
for (i = 0; i < MAX_ARG; i++)

pArgs[i] = argstr[i];
_tprintf (_T("Windows Job Mangement with Windows Job Object.\n")); while
(!exitFlag) {
_tprintf (_T("%s"), _T("JM$"));
_fgetts (command, MAX_COMMAND_LINE, stdin); pc = _tcschr (command, _T('\n'));
*pc = _T('\0');
GetArgs (command, &localArgc, pArgs); CharLower (argstr[0]);

if (_tcscmp (argstr[0], _T("jobbg")) == 0) { Jobbg (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("jobs")) == 0) { Jobs (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("kill")) == 0) { Kill (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("quit")) == 0) { exitFlag = TRUE;
}
else _tprintf (_T("Illegal command. Try again.\n"));
}
CloseHandle (hJobObject);
return 0;
}
```

Topic No.105: (Listing Background jobs)

Job Listing

The job management function used for the purpose is **DisplayJobs()**

The function opens the file. Looks up into the file and acquires the status of the processes listed in the file. Also displays the status of the processes listed and other information.

BOOL DisplayJobs (void)

```
/* Scan the job database file, reporting on the status of all jobs.
        In the process remove all jobs that no longer exist in the system. */
        {
        HANDLE hJobData, hProcess; JM_JOB jobRecord;
        DWORD jobNumber = 0, nXfer, exitCode; TCHAR jobMgtFileName[MAX_PATH];
OVERLAPPED regionStart;
        if ( !GetJobMgtFileName (jobMgtFileName) ) return FALSE;
        hJobData = CreateFile (jobMgtFileName, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hJobData == INVALID_HANDLE_VALUE)
        return FALSE;
        /* Read records and report on each job. */
        /* First, create an exclusive lock on the entire
        file as entries will be changed. */ regionStart.Offset = 0;
        regionStart.OffsetHigh = 0; regionStart.hEvent = (HANDLE)0;
        LockFileEx (hJobData, LOCKFILE_EXCLUSIVE_LOCK, 0, 0, 0, ®ionStart);
          try {
        while (ReadFile (hJobData, &jobRecord, SJM_JOB, &nXfer, NULL) && (nXfer > 0))
{ jobNumber++; /* jobNumber won't exceed MAX_JOBS_ALLOWED as the file is


        not allowed to grow that long */
        hProcess = NULL;
        if (jobRecord.ProcessId == 0) continue;
        /* Obtain process status. */
        hProcess = OpenProcess (PROCESS_ALL_ACCESS, FALSE, jobRecord.ProcessId); if
(hProcess != NULL) {
```

```
GetExitCodeProcess (hProcess, &exitCode); CloseHandle (hProcess);
}
/* Show the job status. */
_tprintf (_T (" [%d] "), jobNumber); if (NULL == hProcess)
_tprintf (_T (" Done")); else if (exitCode != STILL_ACTIVE)
_tprintf (_T ("+ Done"));
else _tprintf (_T ("      "));

/* Show the command. */
_tprintf (_T (" %s\n"), jobRecord.CommandLine);
/* Remove processes that are no longer in the system. */
if (NULL == hProcess) { /* Back up one record. */
/* SetFilePoiner is more convenient here than SetFilePointerEx since the move is
a short shift from the current position */

SetFilePointer (hJobData, -(LONG)nXfer, NULL, FILE_CURRENT);

/* Set the job number to 0. */ jobRecord.ProcessId = 0;
if (!WriteFile (hJobData, &jobRecord, SJM_JOB, &nXfer, NULL)) ReportError (_T
("Rewrite error."), 1, TRUE);
}
} /* End of while. */
} /* End of try. */
 finally {          /* Release the lock on the file. */ UnlockFileEx (hJobData, 0, 0, 0,
®ionStart);
if (NULL != hProcess) CloseHandle (hProcess);
}
CloseHandle (hJobData);
return TRUE;
}
```

Week 10: Topic 106-117

Topic No - 106 (Finding a Process Id)

The job management function used for the purpose is **FindProcessId().**It obtains the process id of the given job number.

It simply looks up into the file based on job number and reads the record at the specific location.

```
DWORD FindProcessId (DWORD jobNumber);
```

```
/* Obtain the process ID of the specified job number. */
{
        HANDLE hJobData;
        JM_JOB jobRecord;
        DWORD nXfer, fileSizeLow;
        TCHAR jobMgtFileName[MAX_PATH+1];
        OVERLAPPED regionStart;
        LARGE_INTEGER fileSize;

        if ( !GetJobMgtFileName (jobMgtFileName) ) return 0;
        hJobData = CreateFile (jobMgtFileName, GENERIC_READ,
                        FILE_SHARE_READ | FILE_SHARE_WRITE,
                        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (hJobData == INVALID_HANDLE_VALUE) return 0;
        /* Position to the correct record, but not past the end of file */
        /* As a variation, use GetFileSize to demonstrate its operation. */
        if (!GetFileSizeEx (hJobData, &fileSize) ||
                (fileSize.HighPart != 0 || SJM_JOB * (jobNumber - 1) > fileSize.LowPart
                 || fileSize.LowPart > SJM_JOB * MAX_JOBS_ALLOWED))
                        return 0;
        fileSizeLow = fileSize.LowPart;
        /* SetFilePoiner is more convenient here than SetFilePointerEx since the the file is known to be
"short" ( < 4 GB). */
        SetFilePointer (hJobData, SJM_JOB * (jobNumber - 1), NULL, FILE_BEGIN);

        /* Get a shared lock on the record. */
        regionStart.Offset = SJM_JOB * (jobNumber - 1);
        regionStart.OffsetHigh = 0; /* Assume a "short" file. */
        regionStart.hEvent = (HANDLE)0;
        LockFileEx (hJobData, 0, 0, SJM_JOB, 0, &regionStart);

        if (!ReadFile (hJobData, &jobRecord, SJM_JOB, &nXfer, NULL))
                ReportError (_T ("JobData file error"), 0, TRUE);

        UnlockFileEx (hJobData, 0, SJM_JOB, 0, &regionStart);
        CloseHandle (hJobData);

        return jobRecord.ProcessId;
}
```

Topic No - 107 (Job Objects)

Windows provides the provision of grouping a number of processes into a job object.

Resource limits can be specified for each job object.

This also helps maintain accounting information.

Creating Job Objects

Firstly a job object is created using **CreateJobObject()**.It uses a name and security attributes. Also, **OpenJobObject()** is used with named object and **CloseHandle()** is used to close the object. Once the job object is created it can be assigned processes using **AssignProcesstoJobObject**().

Creating Job Objects

Once a process is assigned to a job object it cannot be assigned to another job object. Child processes are automatically assigned the same job object as the parent unless the CREATE_BREAKAWAY_FROM_JOB flag is used at creation.

Control limits are specified to the processes in job object using:

```
BOOL SetInformationJobObject(
                    HANDLE hJob,
                    JOBOBJECTINFOCLASS JobObjectInformationClass,
                    LPVOID lpJobObjectInformation,
                    DWORD cbJobObjectInformationLength );
```

1. **hJob** is handled to the existing job object.
2. **JobObjectInformationClass** is the class of the limits you wish to set regarding per process time, working set limit, the limit on active processes, priority, processor affinity, etc.
3. **lpJobObjectInformation** is the structure containning the actual information as it uses a different structure for each class.

Topic No - 108 (Using Job Objects)

In this example job objects are used to limit process execution time and obtain user time statistics.

The simple JobShell program discussed previously is modified.

A command line time limit argument is added.

```
/* Chapter 6 */
/* JobObjectShell.c One program combining three
job management commands:
Jobbg - Run a job in the background.
jobs - List all background jobs.
kill - Terminate a specified job of job family.
 There is an option to generate a console control signal.
This implemenation enhances JobShell with a time limit on each process.
There is a time limit on each process, in seconds, in argv[1] (if present)
0 or omitted means no process time limit
*/

#include "Everything.h"
#include "JobManagement.h"

#define MILLION 1000000
static int Jobbg (int, LPTSTR *, LPTSTR);
static int Jobs  (int, LPTSTR *, LPTSTR);
static int Kill  (int, LPTSTR *, LPTSTR);
HANDLE hJobObject = NULL;

JOBOBJECT_BASIC_LIMIT_INFORMATION basicLimits = {0, 0,
JOB_OBJECT_LIMIT_PROCESS_TIME};

int _tmain (int argc, LPTSTR argv[])
{
```

```
LARGE_INTEGER processTimeLimit;
BOOL exitFlag = FALSE;
TCHAR command[MAX_COMMAND_LINE], *pc;
DWORD i, localArgc;
TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
LPTSTR pArgs[MAX_ARG];
/* NT Only - due to file locking */
if (!WindowsVersionOK (3, 1))
ReportError (_T("This program requires Windows NT 3.1 or greater"), 1,
FALSE);
hJobObject = NULL;
processTimeLimit.QuadPart = 0;
if (argc >= 2) processTimeLimit.QuadPart = _ttoi(argv[1]);
basicLimits.PerProcessUserTimeLimit.QuadPart = processTimeLimit.QuadPart * MILLION;

hJobObject = CreateJobObject(NULL, NULL);
if (NULL == hJobObject)
ReportError(_T("Error creating job object."), 1, TRUE);
if (!SetInformationJobObject(hJobObject, JobObjectBasicLimitInformation, &basicLimits,
sizeof(JOBOBJECT_BASIC_LIMIT_INFORMATION)))
ReportError(_T("Error setting job object information."), 2, TRUE);

for (i = 0; i < MAX_ARG; i++)
pArgs[i] = argstr[i];

_tprintf (_T("Windows Job Mangement with Windows Job Object.\n"));
while (!exitFlag) {
_tprintf (_T("%s"), _T("JM$"));
_fgetts (command, MAX_COMMAND_LINE, stdin);
pc = _tcschr (command, _T('\n'));
*pc = _T('\0');
```

```
GetArgs (command, &localArgc, pArgs);
CharLower (argstr[0]);

if (_tcscmp (argstr[0], _T("jobbg")) == 0) {
Jobbg (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("jobs")) == 0) {
Jobs (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("kill")) == 0) {
Kill (localArgc, pArgs, command);
}
else if (_tcscmp (argstr[0], _T("quit")) == 0) {
exitFlag = TRUE;
}
else _tprintf (_T("Illegal command. Try again.\n"));
}
CloseHandle (hJobObject);
return 0;
}
```

/* Jobbg: Execute a command line in the background, put
the job identity in the user's job file, and exit.
Related commands (jobs, fg, kill, and suspend) can be used to manage the jobs. */

/* jobbg [options] command-line
-**c: Give the new process a console**.
-d: The new process is detached, with no console.
These two options are mutually exclusive.
If neither is set, the background process shares the console with jobbg. */

```
/* These new features this program illustrates:
1. Creating detached and with separate consoles.
2. Maintaining a Jobs/Process list in a shared file.
3. Determining a process status from a process ID.*/
/* Standard includes files. */
/* - - - - - - - - - - - - - - - - - - - - - - - - */

int Jobbg (int argc, LPTSTR argv[], LPTSTR command)
{
/* Similar to timep.c to process command line. */
/* - - - - - - - - - - - - - - - - - - - - - - */
/* Execute the command line (targv) and store the job id,
the process id, and the handle in the jobs file. */
DWORD fCreate;
LONG jobNumber;
BOOL flags[2];
STARTUPINFO startUp;
PROCESS_INFORMATION processInfo;
LPTSTR targv = SkipArg (command, 1, argc, argv);
GetStartupInfo (&startUp);

/* Determine the options. */
Options (argc, argv, _T ("cd"), &flags[0], &flags[1], NULL);

/* Skip over the option field as well, if it exists. */
/* Simplifying assumptions: There's only one of -d, -c (they are mutually exclusive.
   Also, commands can't start with -. etc. You may want to fix this. */
if (argv[1][0] == _T('-'))
targv = SkipArg (command, 2, argc, argv);

fCreate = flags[0] ? CREATE_NEW_CONSOLE : flags[1] ? DETACHED_PROCESS : 0;
/* Create the job/thread suspended.
```

```
Resume it once the job is entered properly. */
if (!CreateProcess (NULL, targv, NULL, NULL, TRUE,
fCreate | CREATE_SUSPENDED | CREATE_NEW_PROCESS_GROUP,
NULL, NULL, &startUp, &processInfo)) {
ReportError (_T ("Error starting process."), 0, TRUE);
}
if (hJobObject != NULL)
{
if (!AssignProcessToJobObject(hJobObject, processInfo.hProcess)) {
ReportError(_T("Could not add process to job object. The process will be terminated."), 0,
TRUE);
TerminateProcess (processInfo.hProcess, 4);
CloseHandle (processInfo.hThread);
CloseHandle (processInfo.hProcess);
return 4;
}
}
```

```
/* Create a job number and enter the process Id and handle
into the Job "database" maintained by the
GetJobNumber function (part of the job management library). */
jobNumber = GetJobNumber (&processInfo, targv);
if (jobNumber >= 0)
ResumeThread (processInfo.hThread);
else {
TerminateProcess (processInfo.hProcess, 3);
CloseHandle (processInfo.hThread);
CloseHandle (processInfo.hProcess);
ReportError (_T ("Error: No room in job control list."), 0, FALSE);
return 5;
}
CloseHandle (processInfo.hThread);
```

```
CloseHandle (processInfo.hProcess);
_tprintf (_T (" [%d] %d\n"), jobNumber, processInfo.dwProcessId);
return 0;
}
/* Jobs: List all running or stopped jobs that have
been created by this user under job management;
that is, have been started with the jobbg command.
Related commands (jobbg and kill) can be used to manage the jobs. */
/* These new features this program illustrates:
1. Determining process status.
2. Maintaining a Jobs/Process list in a shared file.
3. Obtaining job object information
 */
int Jobs (int argc, LPTSTR argv[], LPTSTR command)
{
JOBOBJECT_BASIC_ACCOUNTING_INFORMATION basicInfo;
if (!DisplayJobs ()) return 1;
/* Dispaly the job information */
if (!QueryInformationJobObject(hJobObject, JobObjectBasicAccountingInformation,
&basicInfo, sizeof(JOBOBJECT_BASIC_ACCOUNTING_INFORMATION), NULL)) {
ReportError(_T("Failed QueryInformationJobObject"), 0, TRUE);
return 0;
}
_tprintf (_T("Total Processes: %d, Active: %d, Terminated: %d.\n"),
basicInfo.TotalProcesses, basicInfo.ActiveProcesses, basicInfo.TotalTerminatedProcesses);
_tprintf (_T("User time all processes: %d.%03d\n"),
basicInfo.TotalUserTime.QuadPart / MILLION, (basicInfo.TotalUserTime.QuadPart % MILLION)
/ 10000);

return 0;
}
/* kill [options] jobNumber
```

```
Terminate the process associated with the specified job number. */
/* This new features this program illustrates:
1. Using TerminateProcess
2. Console control events */
/* Options:
-b  Generate a Ctrl-Break
-c  Generate a Ctrl-C
Otherwise, terminate the process. */

/* The Job Management data structures, error codes,
constants, and so on are in the following file. */

int Kill (int argc, LPTSTR argv[], LPTSTR command)
{
DWORD processId, jobNumber, iJobNo;
HANDLE hProcess;
BOOL cntrlC, cntrlB, killed;

iJobNo = Options (argc, argv, _T ("bc"), &cntrlB, &cntrlC, NULL);
/* Find the process ID associated with this job. */

jobNumber = _ttoi (argv[iJobNo]);
processId = FindProcessId (jobNumber);
if (processId == 0) {
ReportError (_T ("Job number not found.\n"), 0, FALSE);
return 1;
}
hProcess = OpenProcess (PROCESS_TERMINATE, FALSE, processId);
if (hProcess == NULL) {
ReportError (_T ("Process already terminated.\n"), 0, FALSE);
return 2;
}
```

```
if (cntrlB)
killed = GenerateConsoleCtrlEvent (CTRL_BREAK_EVENT, 0);
else if (cntrlC)
killed = GenerateConsoleCtrlEvent (CTRL_C_EVENT, 0);
else
killed = TerminateProcess (hProcess, JM_EXIT_CODE);
if (!killed) {
ReportError (_T ("Process termination failed."), 0, TRUE);
return 3;
}
WaitForSingleObject (hProcess, 5000);
CloseHandle (hProcess);
_tprintf (_T ("Job [%d] terminated or timed out\n"), jobNumber);
return 0;
}
```

Topic No - 109 (Thread Overview)

Thread is an independent unit of execution within a process. In a multi-threaded system, multiple threads may exist within a process. Organizing and coordinating these threads is a challenge. Some programming problems are greatly simplified using threads.

The conventional scenario uses single-threaded processes being run concurrently. **This scenario has varying disadvantages.**

1. Switching among processes is time-consuming and expensive. Threads can easily allow concurrent processing of the same function, hence reducing overheads.
2. Usually, independent processes are not tightly coupled, and hence sharing resources is difficult.
3. Synchronization and mutual exclusion in single-threaded processes halt computations.
4. Threads can perform the asynchronously overlapped functions with less programming effort. The use of multithreading can benefit more with a multiprocessor environment using efficient scheduling of threads.

Topic No - 110 (Thread Issues)

Content: Issues with Threads

- ✓ Threads share resources within a process. One thread may inadvertently another thread's data.
- ✓ In some specific cases, concurrency can greatly degrade performance rather than improve.
- ✓ In some simple single-threaded solutions using multithreading greatly complicates matters and even results in poor performance.

Topic No - 111 (Thread Basics)

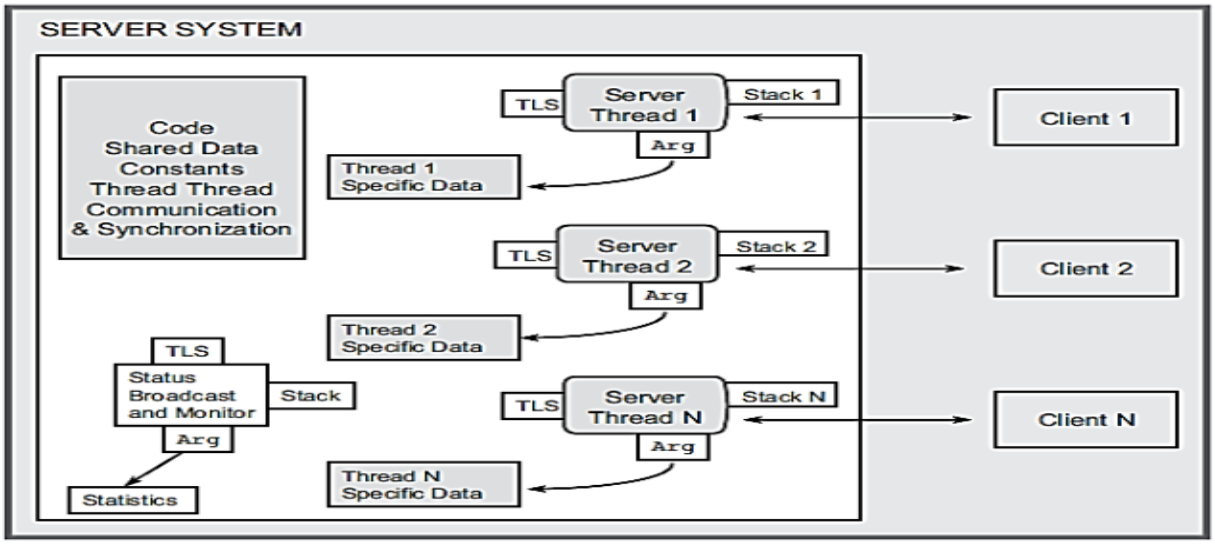
Content: Threads Basics

Threads use the space assigned to a process.

In a multi-threaded system, multiple threads might be associated with a process Thread within a process share data and code. However, individual threads may have their unique data storage in addition to the shared data. This data is not altogether exclusive.

Programmer must assure that a thread only uses its own data and does not interfere with shared data. Moreover, each thread has its own stack for function calls. The calling process usually passes an argument to the thread.

These arguments are stored in the thread's stack Each thread can allocate its own Thread Local Storage (TLS) and set clear values. TLS is a small pointer array that is only accessible to the thread. This also assures that a thread will not modify data of any other thread's TLS.



Topic No - 112 (Thread Management)

Content: Thread Management

Like every other resource threads are also treated as objects. The API used to create a thread is **CreateThread().** Threads can also be treated as parent and child. Although the OS is unaware of that.

CreateThread() has many unique requirements . The **CreateThread()** requires the starting address of the thread within the calling process. It also requires stack space size for the thread which comes from process address space.

Default memory space IMB

- One page is committed in start, this grows with requirement.
- Requires a pointer to thread argument. This can be any structure.

```
HANDLE CreateThread (
                        LPSECURITY_ATTRIBUTES lpsa,
                        SIZE_T dwStackSize,
                        LPTHREAD_START_ROUTINE lpStartAddress,
                        LPVOID lpParameter,
                        DWORD dwCreationFlags,
                        LPDWORD lpThreadId );
```

1. **lpsa** is security attribute's structure used previously.
2. **dwStackSize** is the stack size in bytes.
3. **lpStartAddress** is the starting address of the thread function within the calling process of the form:
4. **DWORD WINAPI ThreadFunc(LPVOID):** The function returns a DWORD value which is usually an exit code and accepts a single pointer argument.
5. **lpThreadParameter** is the pointer passed to the thread and is usually interpreted as a pointer to a structure containing arguments.
6. **dwCreationFlags** if 0 would been that thread would start readily. If its

CREATE_SUSPENDED then the thread will be suspended requiring the use of **ResumeThread ()** to start execution.

7. **lpThreadId** is a pointer to a DWORD that will receive the thread identifier. If its kept NULL then no thread identifier will be returned.

Topic No - 113 (Exiting Thread)

All threads in a process can exit using a **ExitThread()** function. An alternate is that the thread function returns with the exit code. When a thread exits the thread stack is deallocated and the handle referring to the thread are invalidated. If the thread is linked to some DLL then the DllMain() function is invoked with the reason DLL_THREAD_DETACH.

```
VOID ExitThread(DWORD dwExitCode);
```

When all the threads exit the process terminates. One thread can terminate another thread using **TerminateThread()**. In this case Thread resources will not be deallocated, completion handlers do not execute, no notification is sent to attached DLLs. Because of these reasons use of **TerminateThread()** is strongly discouraged. A thread object will continue to exist even its execution has ended until the last reference to the thread handle is destroyed with CloseHandle().

Any other running thread can retrieve the exit code.

```
BOOL GetExitCodeThread (
                    HANDLE hThread,
                    LPDWORD lpdwExitCode);
```

lpdwExitCode contains a pointer to exit code.

The thread is still running the exit code will be STILL_ACTIVE.

Topic No - 114 (Thread Identity)

Thread Ids and handles can be obtained using functions quite similar to one used with processes.

GetCurrentThread()

Non-inheritable pseudohandle to the calling thread.

GetCurrentThreadId()

Obtains thread Id rather than the handle.

GetThreadId()

Obtains thread Id from thread handle.

OpenThread()

Creates Thread handle from the thread Id.

Topic No – 115: (More on Thread Management)

The functions of thread management that are discussed above are enough to program any useful threading application. However, there are some more functions introduced in the later versions of Windows i.e. Windows XP and Windows 2003 to write a useful and robust program. These functions are described below.

GetProcessIdOfThread()

This function was not available in the earlier version of windows and requires Windows 2003 or later versions. When a thread is specified, this function tells us, to which process a specified thread is linked while returning the id of that process. This function is useful for, mapping thread and process and program that manages or interacts with threads in another process.

GetThreadIOPendingFlag()

This function determines whether the thread, specified by its handle, has any outstanding I/O requests. For example, the thread might be blocked for some IO operations. The result is the status at the time that the function is executed; the actual status could change at any time if the target thread completes or initiates an operation.

Suspending and Resuming Threads

In some cases, we might require pausing any running thread or resume any paused thread. For these purposes, Windows maintains a suspend count. Every thread has its separate suspend count. A thread will only run if the suspend count is 0 and if it is not, then the thread is paused, and execution of that thread will be stopped until the suspend count becomes 0. One thread can increment or decrement the suspend count of another thread using **SuspendThread** and **ResumeThread**. Recall that a thread can be created in the suspended state with a count of 1.

```
DWORD ResumeThread (HANDLE hThread)

DWORD SuspendThread (HANDLE hThread)
```

Note: Both functions, if successful, return the previous suspend count. 0xFFFFFFFF indicates failure.

Topic No – 116: (Waiting for Threads)

One thread can wait for another thread to complete or terminate in the same way that threads wait for process termination. Threads and actions should be synchronized so that action can be performed after completing execution. The wait function can be used with the thread handle to check whether a particular thread has completed its execution or not. Window treats thread as an object, and there are two different types of wait function in the windows that can be used for threads as well (Since the thread is also an object), i.e. **WaitForSingleObject()** or **WaitForMultipleObjects(). WaitForSingleObject()** is used to wait for a single specified object and **WaitForMultipleObjects** can be used for more than 1 object, those objects are defined in the form of an array.

For **WaitForMultpleObjects,** there is a defined limit in windows to wait for execution i.e. MAXIMUM_WAIT_OBJECTS(64), Usually, it is 64 objects. If there are more than 64 objects, we can use multiple calls as well i.e. if there are 100 objects, we can create 2 arrays of 64 and 36 objects and call **WaitForMultipleObjects()** two times.

The wait function waits for the object, indicated by the handle, to become signaled. In the case of threads, **ExitThread()** and **TerminateThread()** set the object to the signaled state, releasing all other threads waiting on the object, including threads that might wait in the future after the thread terminates. Once a thread handle is signaled, it never becomes non signaled.

Note that multiple threads can wait on the same object. Similarly, the **ExitProcess()** function sets the process state and the states of all its threads to signaled.

Topic No – 117: (C Library in Threads)

Let's assume a scenario, where we are using the window threading function with C library functions concurrently, there might arise a problem of thread safety. For example, **strtok()** is a C library function, used to extract the token from the specified string, this function uses the global memory space for its internal processing, and if we are using the different copies of that string, they all might use global space.

The result achieved from this operation might get compromised and unsatisfactory. In such cases, these types of problems are resolved by using C Library threading functions rather than Windows threading functions. For C Library, Windows C provides a thread-safe library named LIBCMT. This library can be used for thread-relevant functions to program a multithreaded program.

So far, we were using the CreateThread and ExitThread functions to create and exit the threads, this library provides us with these equivalent functions i.e. _beginthreadex() and endthreadex() respectively. These C library functions are quite simpler but are not diverse as compared to Windows threading functions. i.e._beingthreadex() is a simpler function but it does not allow users to specify the security attributes.

_endthreadex() does not allow return values and does not pass information regarding the status of the thread. If we are using the windows program and using this C Libray thread function the return value must be type cast to HANDLE to process it further, since the original return type of _beginthreadex() is not HANDLE.

Week 11: Topic 118-130

Topic No – 118: (Multithreaded Pattern Searching)

In this example of pattern searching with multithreading, the program is managing concurrent I/O to multiple files, and the main thread, or any other thread, can perform additional processing before waiting for I/O completion. In this way, we can manage several files in a very simple and efficient way. In the previous examples of pattern searching, we used the multitasking approach, but here we will use the multithreading technique, that enables us to write an optimal program.

In Synchronous Input-Output, suppose an example of the keyboard, when the user presses any keyboard button the execution stops until the button is released, but in asynchronous input-output, a Sound card plays the song at the same time other operations are also performed.

When a read operation is performed, this can be performed concurrently, a file can be read by several processes at the same time, or we can also read several files at the same time. But the problem arises when several processes attempt to write a single file. For now, we will be limited to read operation only. This program can be provided several files in which a specific pattern is to be searched. For every file, a separate thread will be created which will search the pattern in that file. Once the pattern is found in the file, it will be reported in a temporary file.

```
/* grepMT. */
/* Parallel grep-- multiple thread version. */

#include "Everything.h"
typedef struct { /* grep thread's data structure. */
        int argc;
        TCHAR targv[4][MAX_PATH];
} GREP_THREAD_ARG;

typedef GREP_THREAD_ARG * PGR_ARGS;
static DWORD WINAPI ThGrep (PGR_ARGS pArgs);

int _tmain(int argc, LPTSTR argv[])
{
        GREP_THREADED_ARG *gArg;
        HANDLE *tHandle;
        DWORD threadIndex, exitCode;
        TCHAR
        commandLine[MAX_COMMAND_LINE)
        ; int iThrd, threadCount;
        STARTUPINFO startUp;
        PROCESS_INFORMATION
        processInfo;

        GetStartupInfo(&startUp);

        /* Boss Thread: create seperate "grep" thread for each file. */
```

```
tHandle = malloc((argc-2) * sizeof(HANDLE));
gArg = malloc((argc-2) * sizeof (GREP_THREAD_ARG));

for(iThrd=0, iThrd<argc-2; iThrd++)
{
_tcscpy (gArg[iThrd].targv[1], argv[1]); /* Pattern */
_tcscpy (gArg[iThrd].targv[2], argv[iThrd + 2]);
GetTempFileName /* Temp file name. */ (".", "Gre", 0, gArg[iThrd].targv[3];
gArg[iThrd].argc = 4;

/* Create a worker thread to execute the command line. */
 tHandle[iThrd] = (HANDLE)_beginthreadex (
                        NULL, 0, ThGrep, &gArg[iThrd], 0, NULL);
}

/* Redirect std Output for file listing process. */ startUp.dwFlags =
STARTF_USESTDHANDLES; startUp.hStdOutput = GetStdHandle
(STD_OUTPUT_HANDLE);

/* Worker threads ar eall running . Wait for them to complete. */ threadCount
= argc - 2;
while (threadCount>0) {
        threadIndex = WaitForMultipleObjects ( threadCount, tHandle, FALSE,
INFINITE);
        iThrd = (int) threadIndex - int (WAIT_OBJECT_0);
        GetExitCodeThread ( tHandle[iThrd], &exitCode);
         CloseHandle ( tHandle[iThrd]);
        if ( exitCode == 0) {   /* Pattern Found */
                if ( argc > 3) {
                        /* Print file name if more than one . */
                        _tprintf ( _T("\n**Search results - file : %s\n"),
                        gArg[iThrd].targv[2];
                        fflush(stdout);
}

/* Use the "cat" program to list the result files. */
         _stprintf ( commandLine, _T("%s%s"), _T("cat "), gArg[iThrd].targv[3]);
         CreateProcess(NULL, commandLine, NULL, NULL, TRUE, 0, NULL, NULL,
         &startUP, &processInfo);
         waitForSingleObject ( processInfo.hProcess, INFINITE); CloseHandle
         (processInfo.hProcess);
         CloseHandle (processInfo.hThread);
}

DeleteFile ( gArg[iThrd].targv[3]);
```

```
            /* Adjust thread and file name arrays. */
            tHandle[iThrd] = tHandle[threadCount - 1];
            _tcscpy(gArg[iThrd].targv[3], gArg[threadCount -1 ].targv[3]);
            _tcscpy(gArg[iThrd].targv[2], gArg[threadCount -1 ].targv[2]);
            threadCount--;
            }
}

/* The form of grep thread function code is :
static  DWORD WINAPI ThGrep (PGR_ARGS pArgs)
{
            ....
} */
```

- Structure of thread argument is created with argument count (argc) and thread argument value (targv) which will be passed to thread. A Thread prototype is also created named as ThGrep which will be used to search the specific patterns in the file.
- STARTUPINFO and PROCESS_INFORMATION structures are created for startup and for creating process respectively.
- stratUp information is placed in GetStratupInfo function.
- Files are specified using the command line, in which patterns are to be searched, and for every file inputted, a separate thread will run.
- Loop will run till argc-2 times, in argc, the first two parameters will be process name and pattern and 3rd parameter is inputted file names, though, this loop will run, till the number of files inputted. In this loop for every file, the name of the file is copied and its temporary file is created
- Further, arguments that are to be passed to the thread are also prepared, 1st argument will store the pattern which is to be searched, 2nd argument will store the name of the file in which the pattern is to be searched and 4th argument will store the count i.e. how many arguments are stored.
- Thread is created using _beginthreadex, whose name is ThGrep and is passed all the arguments that were created (gArg). With this, the handle of every thread will be stored in the form of an array (tHandle)
- Standard output files, standard error files, and flags are set for startup information.
- Total number of threads are stored in ThreadCount
- Another loop is run, in which the Wait function is called to wait for multiple objects specified with a number of threads (threadCount) and handles of all threads (tHandle array). When the wait function is completed, an exitcode will be generated which will tell us why the wait function is stopped, and further, for garbage collection, the handle of the thread is also closed with the CloseHandle function.
- When the wait function was in execution, a process is created with help of the CreateProcess function which has been provided, startup information(startUp), and process information (processInfo).
- Another wait function is also called but this time WaitForSingleObject is called which has been provided the handle of the process named hProcess.

  Once the execution of the process is completed, the handle of the process as well as the thread are closed.

Topic No – 119: (Boss worker and other thread models)

In the example of multithreaded pattern searching, there was one main thread, which was running other threads. Each file was assigned a separate thread, which was finding the pattern in that file. This model is more like a **boss worker model**. The boss worker model is one in which there is one boss and many workers. The Boss assigns work to workers and each worker report result back to the boss. There are many other models, which are used to write an efficient and more understandable multithreaded program depending on the scenarios.

The **work crew model** is one in which the workers cooperate on a single task, each performing a small piece. They might even divide up the work themselves without direction from the boss. Multithreaded programs can employ nearly every management arrangement that humans use to manage concurrent tasks.

The **Client-Server model** is mostly used worldwide, in which a client requests the server, and the server runs a thread for that client. For every client, a separate thread is run at the server end. In this way, the work is done concurrently rather than sequentially. Another major model is the **pipeline model**, where work moves from one thread to the next

There are many advantages to using these models when designing a multithreaded program, including the following.

- Most models can be used which makes things simpler and expedites programming and debugging efforts.
- Models help you obtain the best performance and avoid common mistakes.
- Models naturally correspond to the structure of programming language constructs.
- Maintenance is simplified.
- Troubleshooting is simplified when they are analyzed in terms of a specific model.
- Synchronization and coordination are simplified using well-defined models.

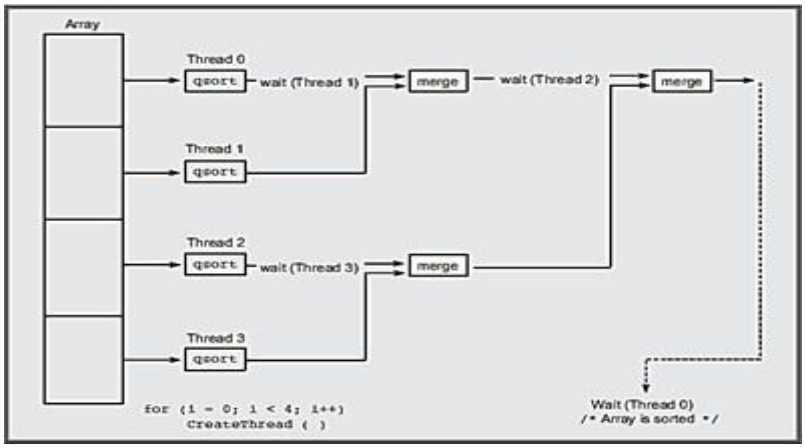
Topic No – 120: (MergeSort: Exploiting Multiple Processors)

The window is a multiprocessing system i.e., more than 1 process can run simultaneously. This example shows, how multithreading can be used to achieve the optimal performance gain in a multiprocessing system, each process runs a different thread to utilize the optimal resources. The main idea behind this is to subdivide the process into similar tasks

so that a separate thread is run for each subtask i.e. A big array is divided into smaller parts, each part is sorted separately with different threads, and all the parts are merged.

This will allow parallelism and better performance gain. The strategy implemented in this example is the worker crew model, work is divided into different workers, and all the work is merged in the end. This strategy could also be used, by using multiprocessing instead of multithreading, but the result might not be as efficient as it is with multithreading, because switching overhead is low in multithreading but high in multiprocessing.

In the example of MergeSort, each subarray is sorted with qsort() and merged as in the mergesort algorithm. The program code will run most accurately if a number of records are divisible by a number of threads and the number of threads is in the power of two. If the number of processes is equal to the number of threads, this would be the most optimal situation otherwise less optimal. If a list is subdivided into 4 sub-lists and 4 threads are created for these sublists, they all must be created at a suspended state and should only be resumed when all the threads are created. If one thread is completed and the other, which is to be merged, is not created, or does not exist, this will occur in a race condition. To avoid the race condition, all the threads should be created with a suspended state and resumed to run all concurrently. The following diagrams explain it further.



A Large array is divided into smaller 4 subarrays. For each subarray, a separate thread is created i.e., thread 0, thread 1, thread 2, and thread 3. When thread 0 is sorted it will wait for thread 1 to be sorted, once sorted, they both will be merged. The same happens for thread 2 and thread 3, they are merged when sorted. These 2 merged subarrays are then sorted and merged to form a large, sorted array.

Topic No – 121: (MergeSort: Exploiting Multiple Processors)

MergeSort: Exploiting Multiple Processors

```
/* Chapter 7 SortMT. Work crew model
        File sorting with multiple threads and merge
        sort. sortMT [options] nt file. Work crew
        model.  */
/*  This program is based on sortHP.
        It allocates a block for the sort file, reads the
        file, then creates an "nt" threads (if 0, use the
        number of processors to so sort a piece of the
        file. It then merges the results in pairs. */
/* LIMITATIONS:
        1.      The number of threads must be a power of 2
        2.      The number of 64-byte records must be a multiple of the number of threads.
        An exercise asks you to remove these
limitations. */

#include "Everything.h"
   /* Definitions of the record structure in the sort file. */
   #define DATALEN 56  /* Correct length for presdnts.txt and monarchs.txt.
   */ #define KEYLEN 8
   typedef struct _RECORD {
           TCHAR
           key[KEYLEN];
           TCHAR
           data[DATALEN];
   } RECORD;
   #define RECSIZE sizeof
   (RECORD) typedef RECORD
   * LPRECORD;
   typedef struct _THREADARG {       /* Thread
           argument */ DWORD iTh;     /* Thread number:
           0, 1, 3, ... */ LPRECORD lowRecord;       /*
           Low Record */ LPRECORD highRecord;  /*
           High record */
   } THREADARG, *PTHREADARG;
   static DWORD WINAPI SortThread (PTHREADARG
   pThArg); static int KeyCompare (LPCTSTR, LPCTSTR);
   static DWORD nRec; /* Total number of records to be
   sorted. */ static HANDLE * pThreadHandle;
   int _tmain (int argc, LPTSTR argv[])
   {

           /* The file is the first argument. Sorting is done in place. */
           /* Sorting is done in memory heaps. */
           HANDLE hFile, mHandle; LPRECORD pRecords = NULL;
           DWORD lowRecordNum, nRecTh,numFiles, iTh;
           LARGE_INTEGER fileSize;
           BOOL noPrint;
           int iFF, iNP;
```

```
PTHREADARG threadArg;
LPTSTR stringEnd;
iNP = Options (argc, argv, _T("n"), &noPrint, NULL);
 iFF = iNP + 1;
numFiles = _ttoi(argv[iNP]);
 if (argc <= iFF)
        ReportError (_T ("Usage: sortMT [options] nTh files."), 1, FALSE);
/* Open the file and map it */
hFile = CreateFile (argv[iFF], GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
if (hFile == INVALID_HANDLE_VALUE)
        ReportError (_T ("Failure to open input file."), 2, TRUE);
// For technical reasons, we need to add bytes to the end.
/* SetFilePointer is convenient as it's a short addition from the file end */
 if (!SetFilePointer(hFile, 2, 0, FILE_END) || !SetEndOfFile(hFile))
        ReportError (_T ("Failure position extend input file."), 3, TRUE);

mHandle = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
if (NULL == mHandle)
        ReportError (_T ("Failure to create mapping handle on input file."), 4, TRUE);

/* Get the file size. */
if (!GetFileSizeEx (hFile, &fileSize))
        ReportError (_T ("Error getting file size."), 5, TRUE);

nRec = (DWORD)fileSize.QuadPart / RECSIZE;/* Total number of records. Note assumed limit */
nRecTh = nRec / numFiles;         /* Records per thread. */
threadArg = malloc (numFiles * sizeof (THREADARG)); /* Array of thread args. */ pThreadHandle
= malloc (numFiles * sizeof (HANDLE));

/* Map the entire file */
pRecords = MapViewOfFile(mHandle, FILE_MAP_ALL_ACCESS, 0, 0, 0);

if (NULL == pRecords)
        ReportError (_T ("Failure to map input file."), 6, TRUE); CloseHandle (mHandle);
/* Create the sorting threads. */ lowRecordNum = 0;
for (iTh = 0; iTh < numFiles; iTh++) {
        threadArg[iTh].iTh = iTh;
        threadArg[iTh].lowRecord = pRecords + lowRecordNum;
         threadArg[iTh].highRecord = pRecords + (lowRecordNum + nRecTh); lowRecordNum +=
        nRecTh;
         pThreadHandle[iTh] = (HANDLE)_beginthreadex ( NULL, 0, SortThread, &threadArg[iTh],
        CREATE_SUSPENDED, NULL);
```

```
/* Resume all the initially suspened threads. */ for (iTh = 0; iTh < numFiles; iTh++)
ResumeThread (pThreadHandle[iTh]);
/* Wait for the sort-merge threads to complete. */ WaitForSingleObject (pThreadHandle[0], INFINITE); for (iTh = 0; iTh < numFiles; iTh++)
CloseHandle (pThreadHandle[iTh]);

/* Print out the entire sorted file. Treat it as one single string. */ stringEnd = (LPTSTR) pRecords + nRec*RECSIZE;
*stringEnd =_T('\0'); if (!noPrint) {
_tprintf (_T("%s"), (LPCTSTR) pRecords);
}
UnmapViewOfFile(pRecords);
// Restore the file length
/* SetFilePointer is convenient as it's a short addition from the file end */ if (!SetFilePointer(hFile, -2, 0, FILE_END) || !SetEndOfFile(hFile))

ReportError (_T("Failure restore input file lenght."), 7, TRUE); CloseHandle(hFile);
free (threadArg); free (pThreadHandle); return 0;
} /* End of _tmain. */
static VOID MergeArrays (LPRECORD, DWORD); DWORD WINAPI SortThread (PTHREADARG pThArg)
{

ReportError (_T("Failure restore input file lenght."), 7, TRUE); CloseHandle(hFile);
free (threadArg); free (pThreadHandle); return 0;
} /* End of _tmain. */
static VOID MergeArrays (LPRECORD, DWORD); DWORD WINAPI SortThread (PTHREADARG pThArg)
{

DWORD groupSize = 2, myNumber, twoToI = 1;
/* twoToI = 2^i, where i is the merge step number. */ DWORD_PTR numbersInGroup;
LPRECORD first; myNumber = pThArg->iTh; first = pThArg->lowRecord;
numbersInGroup = (DWORD)(pThArg->highRecord - first);
```

```
/* Sort this portion of the array. */
qsort (first, numbersInGroup, RECSIZE, KeyCompare);
/* Either exit the thread or wait for the adjoining thread. */
while ((myNumber % groupSize) == 0 && numbersInGroup < nRec) {
/* Merge with the adjacent sorted array. */
WaitForSingleObject (pThreadHandle[myNumber + twoToI], INFINITE);
MergeArrays (first, numbersInGroup);
numbersInGroup *= 2;
groupSize *= 2;  twoToI *=2;
}
return 0;
}
static VOID MergeArrays (LPRECORD p1, DWORD nRecs)
{
/* Merge two adjacent arrays, each with nRecs records. p1 identifies the first */
DWORD iRec = 0, i1 = 0, i2 = 0;
LPRECORD pDest, p1Hold, pDestHold, p2 = p1 + nRecs;
pDest = pDestHold = malloc (2 * nRecs * RECSIZE);
p1Hold = p1;
while (i1 < nRecs && i2 < nRecs) {
if (KeyCompare ((LPCTSTR)p1, (LPCTSTR)p2) <= 0) {
memcpy (pDest, p1, RECSIZE);
i1++; p1++; pDest++;
}
else {
memcpy (pDest, p2, RECSIZE);
i2++; p2++; pDest++;
}
}
if (i1 >= nRecs)
memcpy (pDest, p2, RECSIZE * (nRecs - i2));
else memcpy (pDest, p1, RECSIZE * (nRecs - i1));
memcpy (p1Hold, pDestHold, 2 * nRecs * RECSIZE);
free (pDestHold);
return;
}
int KeyCompare (LPCTSTR pRec1, LPCTSTR pRec2)
{
DWORD i; TCHAR b1, b2;
LPRECORD p1, p2;
int Result = 0;
p1 = (LPRECORD)pRec1;
p2 = (LPRECORD)pRec2;
for (i = 0; i < KEYLEN && Result == 0; i++) {
b1 = p1->key[i];
b2 = p2->key[i];
if (b1 < b2) Result = -1;
if (b1 > b2) Result = +1;
}
return Result;
}
```

- A thread structure is defined which is passed to every thread as an argument, this structure has the 'iTh' variable which specifies the thread number, 'lowRecord' and 'highRecord' which are used to define the starting and ending index of the sub list.
- 'SortThread' is the thread name that is to be created
- 'nRec' is the total number of threads created
- 'pThreadHandle' is the handle to the thread
- The file, which has records in it, is opened to read and write. This file is mapped ('mHandle') to access that file as memory i.e. in the form of an array.
- Thread argument, which is to be passed as an argument to create a thread, is allocated the memory, that has all three fields that are to be passed
- 'nRec' is the total number of records
- 'nRec/numFiles' defines the number of records per file
- 'pRecord' is the address of the mapped file
- Different threads are created and run to divided the file into chunks and each thread sorts the file. To divide the file, a loop is run till 'numFiles' times, and specifies the thread numbers ('ith'), starting index of a chunk ('lowRecord'), and ending index of a chunk ('highRecord')
- The thread is created with 'beginThread' functions which are passed 'SortThread' and 'ThreadArgument' in a suspended state.
- When all the threads are created, they are resumed with help of a loop.
- For all created threads, we wait till the execution completes. Once the execution is completed handle is closed and the file is unmapped.
- When a thread is run, it is passed 'QSort' to sort the records, and then it waits for its adjacent thread to be sorted and completed. Once they are completed, the results are merged. Every thread waits for its adjacent thread to be completed and merges after completion.

Performance

- Multithreading gives the best results when the number of processors is the same as the number of threads.
- Additional threads beyond processor count slows down the program
- Performance degrades if there is one processor and memory is low and the array is very large because most of the time threads will be contending for physical memory. But the problem alleviates when the memory is sufficient.

Topic No – 122: (Introduction to Parallelism)

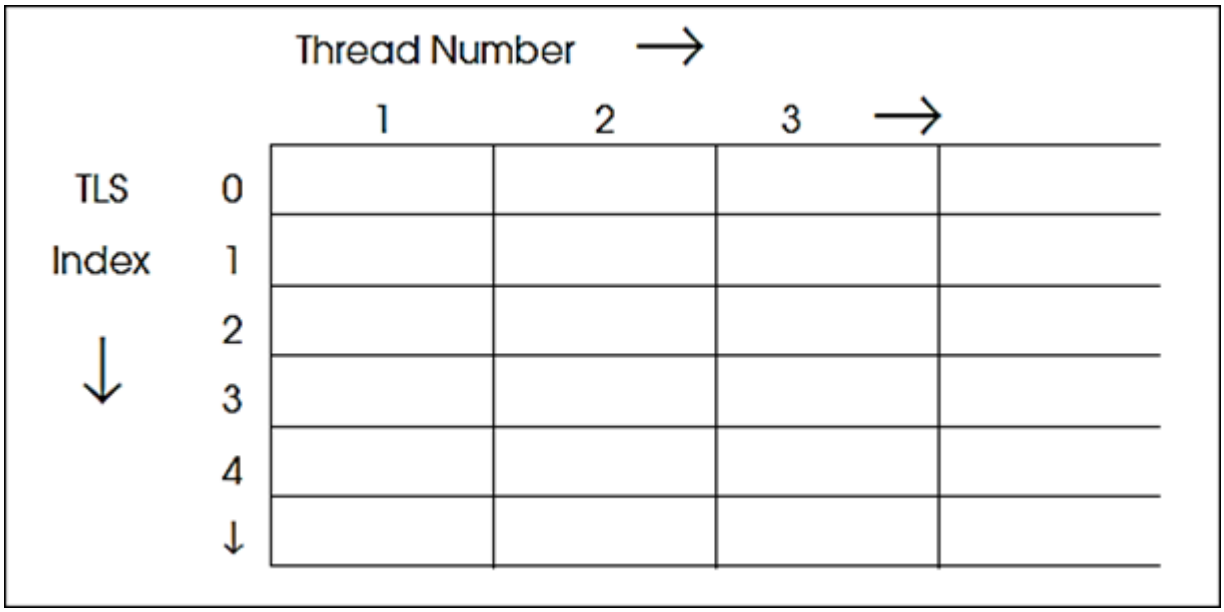
Multiprocessing and **multithreading**, both are responsible for multiple flows of execution, but the multithreading is optimal. Windows is not only multithreading or multiprocessing it also supports multiprocessors as well. If we have a system with multiprocessors, we should learn to program to use the potential of multiprocessors because the processor's speed has reached its bottleneck i.e. after a certain limit, its speed cannot be enhanced.

Parallelization is the key to future performance improvement since we can no longer depend on increased CPU clock rates and since multicore and multiprocessor systems are increasingly common. Previously, various programs have been discussed that unleash the power of parallelism. The properties that enabled parallelism include the following:

- Major task is divided into subtasks and many worker threads were run. Subtasks were divided into worker threads that perform their work. These worker subtasks run independently, without any interaction between them.
- As subtasks are complete, a master program can merge the results of divided subtasks into a single result.
- The programs do not require mutual exclusion of any sort. Only the master worker is synchronized with each worker and waits for them to complete.
- Every worker will work as a separate thread on a separate processor. it is the most optimal situation.
- Program performance scales automatically, up to some limit, as you run on systems with more processors; the programs themselves do not, in general, determine the processor count on the host computer. Instead, the Windows kernel assigns worker subtasks to available processors.
- Output remains undisturbed even if the program is serialized.
- If you "serialize" the program the results should get precisely the same as the parallel program. The serialized program is, moreover, much easier to debug.
- The maximum performance improvement is limited by the program's "**parallelism**," thread management overhead, and computations that cannot be parallelized.

Topic No – 123: (Thread Local Storage)

A thread is an execution unit. In a multithreading program, one procedure can have several threads. Every thread needs data, that it doesn't want to share with other threads, and which is unique i.e. it varies from thread to thread. One technique is to have the creating thread call CreateThread (or beginThreadex) with lpvThreadParm pointing to a data structure that is unique for each thread. The thread can then allocate additional data structures and access them through lpvThreadParm. Windows also provides Thread Local Storage (TLS), which gives each thread its array of pointers. The following figure shows this TLS arrangement.



A function can have many threads i.e., thread 1, thread 2, etc. Every column in the TLS arrangement corresponds to thread numbers and every thread needs variables i.e., TLS index 0,1,2,3 etc. Initially, no TLS indexes (rows) are allocated, but new rows can be allocated and deallocated at any time. Once the row is allocated, it will be allocated to all rows. The primary thread would be a logical choice for TLS space management; however, every thread can access TLS.

```
DWORD TlsAlloc(VOID):
```

- This API is used to allocate the index and it returns the TLS index in the form of the double word. **Otherwise returns -1 in case of failure.**

```
BOOL TlsFree (DWORD dwIndex);
```

The above API Frees the specified index.

```
LPVOID TlsGetValue (DWORD dwTlsIndex )
```

The above Api Provided valid indexes are used. The programmer can access TLS space using these simple GET/SET APIs

Some Cautions

- TLS provides a convenient mechanism for accessing memory that is global within a thread but inaccessible to other threads.
- Global storage of a program is accessible by all threads

Topic No – 124: (Processes and Thread priorities)

In a multitasking or multithreading system, there are a number of processes running, each of which competes for resources like processor, memory, etc. The operating system is responsible for managing the resources. The Windows kernel always runs the highest-priority thread that is ready for execution.

A thread is not ready if it is waiting, suspended, or blocked for some reason. Since the threads are dependents on the processes and they receive priority relative to their process priority classes. Process priority classes are set initially when they are created using **CreateProcess**, and each has a base priority, with values including.

- **IDLE_PRIORTY_CLASS** for threads that will run only when the system is idle. This is the lowest priority process.
- **NORMAL_PRIORTY_CLASS** indicating no special scheduling requirements.
- **HIGH_PRIORTY_CLASS** indicating time-critical tasks that should be executed immediately.
- **REALTIME_PRIORTY_CLASS**, the highest possible priority

The priority class of a process can be set and got using these API's

```
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority)

DWORD GetPriorityClass(HANDLE hProcess)
```

- There are some enhances variants of priority levels i.e. **ABOVE_NORMAL_PRIORTY_CLASS** (which is below HIGH_PRIORTY_CLASS) and BELOW_NORMAL_PRIORTY_CLASS (which is above IDLE_PRIOTY_CLASS).
- **PROCESS_MODE_BACKGROUND_BEGIN**, which lowers the priority of the process and its threads for background work without affecting the responsiveness of foreground processes and threads.
- **PROCESS_MODE_BACKGROUND_END** restores the process priority to the value before it was set with PROCESS_MODE_BACKGROUND_BEGIN.

Thread priorities are either absolute or are set relative to the process base priority. At thread creation time, the priority is set to that of the process. The relative thread priorities are in a range of ±2 "points" from the process's base. The symbolic names of the resulting common thread priorities, starting with the five relative priorities, are:

- THREAD_PRIORTY_LOWEST
- THREAD_PRIORTY_BELOW_NORMAL
- THREAD_PRIORTY_NORMAL
- THREAD_PRIORTY_ABOVE_NORMAL
- THREAD_PRIORTY_HIGHEST
- THREAD_PRIORTY_TIME_CRITICAL is 15, or 31 if the process class is REAL_TIME_PRIORTY_CLASS
- THREAD_PRIORTY_IDLE is 1, or 16 for REAL_TIME_PRIORTY_CLASSES processes.

THREAD_MODE_BACKGROUND_BEGIN and **THREAD_MODE_BACKGROUND_END** are similar to PROCESS_MODE_BACKGROUND_BEGIN and PROCESS_MODE_BACKGROUND_END.

Thread priorities can be set within the range of +-2 of the corresponding process priority using these.

BOOL SetThreadPriority (HANDLE hThread, int nPriority)

Int GetThreadPriority (HANDLE hThread).

Thread priorities are dynamic. They change with the priority of process or windows may also boost thread priority as per need. This feature can be enabled disabled using **SetThreadPriorityBoost().**

Topic No – 125: (Thread States)

The following figure shows how the executive manages threads and shows the possible thread states. This figure also shows the effect of program actions. Such state diagrams are common to all multitasking OSs and help clarify how a thread is scheduled for execution and how a thread moves from one state to another.

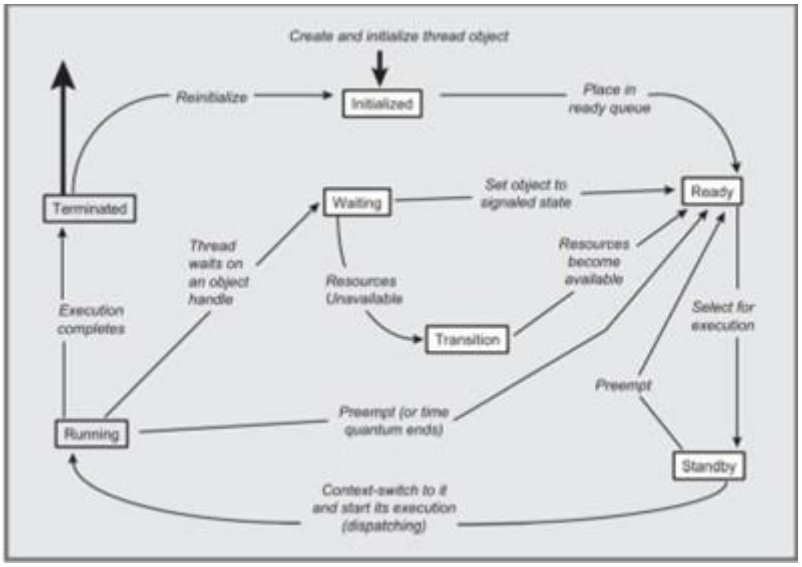


Figure 1 Thread States and Transitions

- A thread is in the running state when it is running on a processor. More than one thread can be in the running state on a multiprocessor computer.
- The executive places a running thread in the wait state when the thread performs a wait on a non-signaled handle, such as a thread or process handle. I/O operations will also

wait for the completion of a disk or other data transfer, and numerous other functions can cause waiting. It is common to say that a thread is blocked, or sleeping, when in the wait state

- A thread is ready if it could be running. The executive's scheduler could put it in the running state at any time. The scheduler will run the highest-priority ready thread when a processor becomes available, and it will run the one that has been in the ready state for the longest time if several threads have the same high priority. The thread moves through the standby state before entering the ready state.
- The executive will move a running thread to the ready state if the thread's time slice expires without the thread waiting. Executing will also move a thread from the running state to the ready state.
- The executive will place awaiting thread in the ready state as soon as the appropriate handles are signaled, although the thread goes through an intermediate transition state. It is common to say that the thread wakes up.
- A thread, regardless of its state, can be suspended, and a ready thread will not be run if it is suspended. If a running thread is suspended, either by itself or by a thread on a different processor, it is placed in the ready state.
- A thread is in the terminated state after it terminates and remains there as long as there are any open handles on the thread. This arrangement allows other threads to interrogate the thread's state and exit code.
- Normally, the scheduler will place a ready thread on any available processor. The programmer can specify a thread's processor, which will limit the processors that can run that specific thread. In this way, the programmer can allocate processors to threads and prevent other threads from using these processors, helping to assure responsiveness for some threads. The appropriate functions are SetProcessAffinityMask and GetProcessAffinityMas. SetThreadIdealProcessor can specify a preferred processor that the scheduler will use whenever possible; this is less restrictive than assigning a thread to a single processor with the affinity mask.

Topic No – 126: (Mistakes while using Threads)

There are several factors to keep in mind as you develop threaded programs; lack of attention to a few basic principles can result in serious defects, and it is best to avoid the problems in the first place than try to find them during testing or debugging.

- The essential factor is that the threads execute asynchronously.
- There is no sequencing unless you create it explicitly.
- This asynchronous behavior is what makes threads so useful, but without proper care, serious difficulties can occur.

Here are a few guidelines. There may be a few inadvertent violations, however, which illustrates the multithreaded programming challenges.

- Make no assumptions about the order in which the parent and child threads execute.
- It is possible for a child thread to run to completion before the parent, or, conversely, the child thread may not run at all for a considerable period.
- On a multiprocessor computer, the parent and one or more children may even run concurrently.
- Make sure all the initializations required by a child thread have been performed before calling **CreateThread().**
- In case a thread has been run but initialization is required then use some technique like thread suspension until data is initialized.
- Failure by the parent to initialize data required by the child is a common cause of “race conditions” wherein the parent “races” the child to initialize data before the child needs it.
- Any thread, at any time, can be preempted, and any thread, at any time, may resume execution.
- Do not confuse synchronization and priority. These both are different concepts. Threads are defined as their priorities when created, However Threads are synchronized in such a way that one thread completes its specific purpose, and only then another thread may run its process.
- Even more so than with single-threaded programs, testing is necessary, but not

sufficient, to ensure program correctness. It is common for a multithreaded program to pass extensive tests despite code defects. There is no substitute for careful design, implementation, and code inspection.

- Threaded program behavior varies widely with processor speed, number of processors, OS version, and more. Testing on a variety of systems can isolate numerous defects, but the preceding precaution still applies.
- The default stack size for a thread is 1MB. Make sure the size is sufficient as per thread needs.
- Threads should be used only as appropriate. Thus, if there are activities that are naturally concurrent, each such activity can be represented by a thread. If, on the other hand, the activities are naturally sequential, threads only add complexity and performance overhead.
- If you use a large number of threads, be careful, as the numerous stacks will consume virtual memory space and thread context switching may become expensive. In other cases, it could mean more threads than the number of processors.
- Fortunately, correct programs are frequently the simplest and have the most elegant designs. Avoid complexity wherever possible.

Topic No – 127: (Timed Waits)

In threading, there are some functions that can be used to wait for threads. The Sleep function allows a thread to give up the processor and move from the running to the wait state for a specified period of time. After the completion of the specified time, the thread will move to the ready state and will wait to move on running state. A thread can perform a task periodically by sleeping after carrying out the task.

```
VOID Sleep (DWORD dwMilliseconds)
```

The time period is specified in milliseconds and can even be INFINITE, in which case the thread will never resume. A 0 value will cause the thread to relinquish the remainder of the time slice; the kernel moves the thread from the running state to the ready state.

The function SwitchToThread() provides another way for a thread to yield its processor to another ready thread if there is one that is ready to run.

Topic No – 128: (Fibers)

A **fiber**, as the name implies, is a piece of a thread. More precisely, **fiber** is a unit of execution that can be scheduled by the application rather than by the operating system. An application can create numerous fibers, and the fibers themselves determine which fiber will execute next. The **fibers** have independent stacks but otherwise run entirely in the context of the thread on which they are scheduled, having access, for example, to the thread's TLS and any mutexes owned by the thread.

Furthermore, fiber management occurs entirely in user space outside the kernel. Fibers can be thought of as lightweight threads, although there are numerous differences. A fiber can execute on any thread, but never on two at one time. Fiber that is meant to run on different threads at different instances should not access thread-specific data from TLS.

Fiber Uses

- Fibers make portability easy.
- Fibers need not wait/block on file locks, mutexes, and pipes. They can pool resources and in case resources are not available they can switch to another fiber.
- Fiber exhibits flexibility. They exist as part of threads, but they are not bound to any specific thread. They can run on any thread but only one at a time.
- Fibers are not pre-emptively scheduled. Windows OS is unaware of fibers. They are controlled and managed by fiber DLL.
- Fibers can be used as co-routines. Using this an application can switch between related tasks. Whereas, in the case of threads applications has no control over which thread executes next.
- Major software platform offers the use of fibers and claims its performance advantages.

Topic No – 129: (Fiber APIs)

A set of different API functions are provided that can help create and manage fibers. These functions are.

A thread must enable fiber operation by calling.

```
ConvertThreadToFiber()
Or
ConvertThreadToFiberEx()
```

After calling this API the thread will now contain a fiber. It will provide a pointer to the fiber

data more or less like thread data. This can be used accordingly.

Subsequently, new fibers can be created in this thread using.

```
CreateFiber()
```

Each new fiber has a start address, stack size, and a parameter. Each fiber is identified by address and not a handle.

Individual fiber can obtain their data by calling.

```
GetFiberData()
```

Similarly, a fiber can obtain its identity by calling.

```
GetCurrentFiber()
```

A running fiber gives control to another fiber using.

```
SwitchToFiber()
```

It uses the address of the other fiber. The context of the current fiber is saved, and the context of the other fiber is restored. Fibers must explicitly indicate the next fiber that is to run in the application.

A running fiber gives control to another fiber using.

```
SwitchToFiber()
```

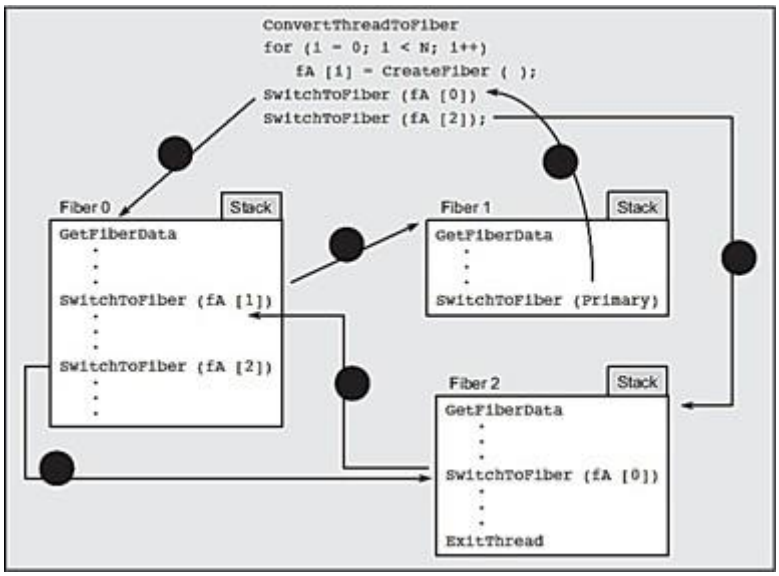
It uses the address of the other fiber. The context of the current fiber is saved, and the context of the other fiber is restored. Fibers must explicitly indicate the next fiber that is to run in the application.

An existing fiber can be deleted using.

```
DeleteFiber()
```

Topic No – 130: (Using Fibers)

Previously we discussed several APIs used to manage the fibers, here we will develop a scheme with these APIs to use the fibers. Fiber enables us to control the switching of threads. This scheme firstly converts a thread to fiber and then uses it as the primary fiber. The primary fiber creates other fibers and switching among these fibers is managed by the application.



In the center top, a primary thread is created in which **ConvertThreadToFiber** is used to convert into fiber and then with the help of a loop number of fibers are created. To convert the execution to a certain fiber **SwitchToFiber** is used. Primary fiber is switched to Fiber 0 which gets fiber data and then switches to Fiber 1 which also performs the same work i.e. gets data and this fiber switches to primary fiber. The primary fiber starts execution from the point where it was switched before i.e., now it switches to fiber 2 which gets data and switches to fiber 0 and then back to fiber 2 and at the end Thread is closed.

There are two policies.

- **Master-Slave Scheduling:** One fiber decides which fiber to run. Each fiber transfers back the execution to the primary fiber. (Fiber 1)
- **Peer to Peer Scheduling:** A fiber determine which fiber should be next to execute based on some policy (Fiber 0 and 2)

Week 12: Topic 131-141

Topic No – 131: (Need for Thread Synchronization)

Threads enable concurrent processing and parallelism in multiple processors. However, there are some pros and cons because of this concurrent processing. When many threads may run concurrently. They may need to synchronize for tasks in numerous instances. i.e. in the Boss-worker model, there are one boss thread and many workers threads, the Boss thread waits for all workers to complete execution and compiles all the data.

In case if boss executes before the worker completes its execution, this may affect the output we are required to ensure that the Boss will not access the worker's memory unless the worker completes it.

Alternately, the workers do not start working unless the boss has created all the workers so that a worker may not try to access the data of another worker which has not been created as yet. When multiple threads are using share data, they may require coordination i.e. when a thread is using the data other threads should wait for it.

Also, programmers need to ensure that two or more threads are not modifying the same data item simultaneously. In case many threads are modifying elements of the queue, the programmer needs to assure that two or more threads do not attempt to remove an element at the same time. Several programming flaws can leave such vulnerabilities in the code.

Example

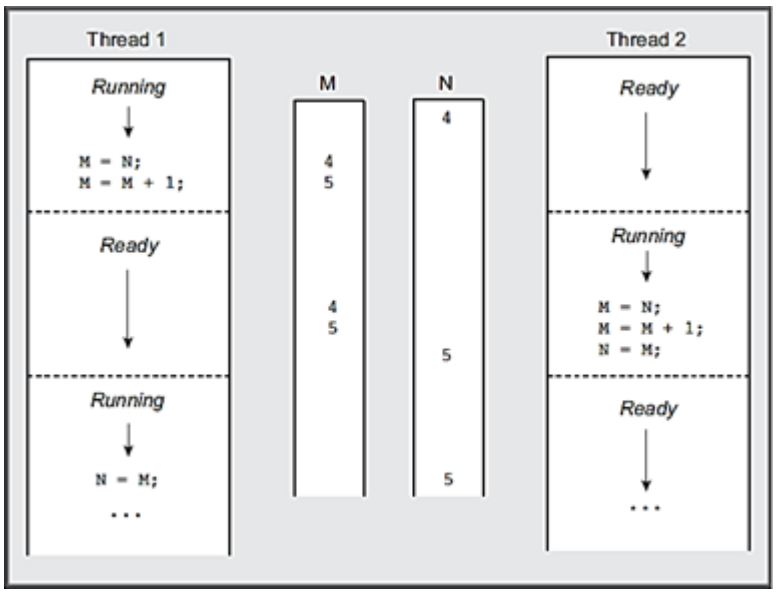
There are two threads i.e. Thread 1 and Thread 2, operation of both of these is the same i.e.

```
Thread 1
{
M = N
M = M + 1
N = N
}
```

```
Thread 2
{
M = N
M = M +
1
N = N
}
```

Let's suppose that the initial value of N is 4 though the final values after execution of thread 1 will be M=5, N=5. After completion of thread 1 when thread 2 starts executing, the initial value of N is 5 (because of thread 1 execution) thus final values become M=6 and N=6.

When these same threads are run concurrently as shown in the following figure.



When the thread is executing, the value of N is 4 initially. After two instructions thread is switched. Thus, the value of M becomes 5 and the value of N is not changed because 3rd instruction is not executed yet. Since the thread is switched therefore thread 2 starts execution and value N is 4 here. It executes all three instructions and values become M=5 and N=5. Again, thread is switched and the remaining 1 instruction of thread 1 executes which affects the N variable and because of this final value of N becomes 5.

Final Values in normal processing M=6 and N=6

Final Values in concurrent processing M=5 and N=5

Thus, we can see in many situations the final output of concurrent processing may not be same as it is in the normal processing.

Topic No – 132: (A Simple Solution to Critical Section Problem)

When problems and errors arise because of parallelism or concurrent processing these types of errors are of critical section problem i.e. A critical resource which is used by number of processes or threads. This critical section problem can be handled by the following ways:

Using Global Variables

This solution uses a global variable named Flag to indicate to all threads that a thread is modifying a variable. A thread turns it TRUE before modifying a variable turns it to FALSE after modifying it. Each thread would check this Flag before modifying the variable. If the Flag is TRUE, then if indicates that the variable is being used by some other thread.

```
BOOL Flag = FALSE;
DWORD N;
. . .
DWORD WINAPI ThreadFunc(TH_ARGS pArgs)
{
   . . .
   while (Flag) Sleep (1000);
   Flag = TRUE;
   N++;
   Flag = FALSE;
   . . .
}
```

Even in this case, the thread could be preempted between the time FLAG is tested and the time FLAG is set to TRUE; the first two statements form a critical code region that is not properly protected from concurrent access by two or more threads. Another attempted solution to the critical code region synchronization problem might be to give each thread its own copy of the variable, as follows:

```
DWORD WINAPI ThreadFunc (TH_ARGS pArgs)
{  DWORD N;
   ... N++; ...
}
```

Topic No – 133: (Volatile Storage)

The volatile stage is a Windows level or Compiler level facility provided when using incrementing operations which change the shared variables to reduce the conflicts that arise because of switching.

Latent Defects

Even if the synchronization problem is somehow resolved there still may remain some latent problems. A thread code may switch to another thread while a variable value has been modified in a register without writing it back. The use of registers for intermediate operations is a compiler optimization technique.

Turning off optimization may adversely affect the performance of the whole program but not always sometimes it may slow the program. ANSI C provides a qualifier volatile for this purpose. A variable with a volatile qualifier will always be accessed from memory for operations and will always be stored in memory after any operation. A volatile qualifier also means that the variable can be accessed at any instance. A volatile qualifier should only be used where necessary as it degrades the performance.

Using volatile

- Use volatile only if a variable is being modified by two or more threads.
- Even if it's read only for two or more threads but the outcome of the threads depends on its new value.

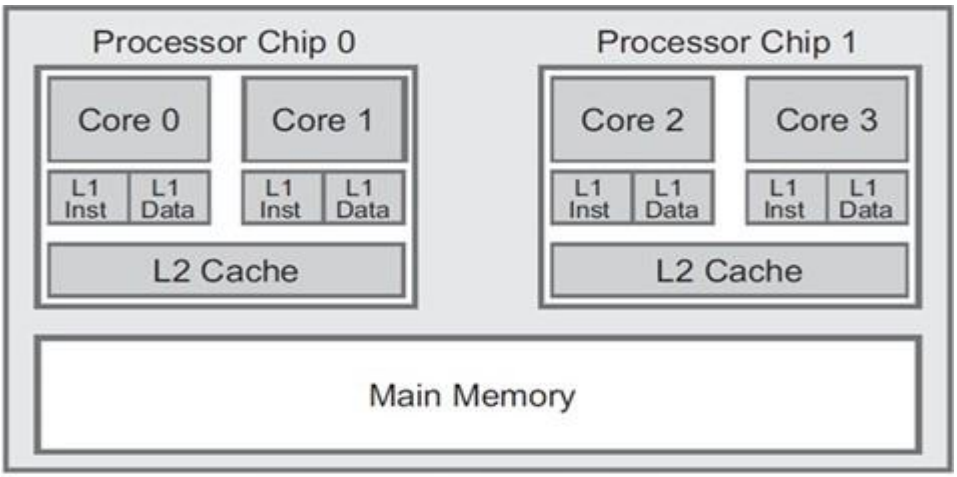
Topic 134: (Memory Architecture and Memory Barriers)

Cache Coherency:

The volatile qualifier does not assure that the changes are visible to processors in a desired order. Processor usually hold the values in cache before writing them back to memory. This may alter the sequence in which different processors see the values.

Memory Barriers:

To assure that the memory is accesses in the desired order use memory barrier or memory fences. The interlocked function provides memory barrier. Further, the concept is clarified the diagram showing 4 processors in a system on two dual core chips.



Memory System Architecture

The diagram depicts 4 processor cores on two chips. Each core has its own register with intermediate values of variables. Each core has separate Level-1 cache for instruction and data. A common larger L2 cache for cores on each chip. Memory is shared among cores.

Volatile qualifiers only assures that the new data values will be updated in L1. There is no guarantee that the values will be visible to other thread running on different processors. Memory barriers assure that the main memory is updated, and cache of all processors is coherent.

For example, if core 0 updates a variable N using a memory barrier. If core 3 L1 cache also contains the values of N then the value in its cache is either updated or removed so that core 3 could access the new value coherent value of N. This certainly incurs a huge cost. Moving data within the core registers costs less than a cycle whereas more data from one core to another via main memory can cost 100s of cycles.

Topic No – 135: (Interlocked Functions

Interlocked functions are most suited if variable with volatile scope only need to be incremented, decremented and exchanges. Interlocked functions are simpler and faster and easy to use. However, they do pose the performance drawback as they do generate a memory barrier.

The most basic of these functions **are InterlockedIncrement()** and **InterlockedDecrement().**

```
Long InterlockedIncrement(LONG volatile *Addend)


Long InterlockedDecrement(LONG volatile *Addend)
```

They both use a 32-bit signed variable as parameter that should be stored at 4-byte boundary in memory. They should be used whenever possible to improve performance.

For 64-bit systems InterlockedIncrement64() and InterlockedDecrement64() can be used with Addend placed at 8-byte boundary.

The function can be used in following way: **InterlockedIncrement(&N);**

N should be volatile integer placed on appropriate memory boundary. Function returns the new value of N. However, some other thread may preempt before the value is returned and change the value of N. Do not call it twice to increment the value by two as the thread may be preempted between both calls. Rather use **InterlockedExchangeAdd**().

Topic No – 136: (Local and Global Storage)

Another criterion for correct thread code is that global storage should not be used for local storage purpose. If a global variable is used to store thread specific data then it will be used by other threads as well. This will result in incorrect behavior no matter how the rest of code is written. The following example is incorrect usage of the global variables.

```
DWORD N;
....
DWORD WINAPI ThreadFunc (TH_ARGS pArgs)
{
...
N = 2 * pArgs->Count; ...
/* Use N; value is specific to this call to ThreadFunc*/
```

```
}
```

N is kept global but is used to store thread specific information from its parameters structure. For example, if many thread functions are running at the same time and each function modifies N with its own parameters, no function will be able to use N properly because N is global, and all threads are running at the same time and affecting N's value. Due to this problem, the final results will be incorrect.

It's important to know when to use local and when to use global variables while dealing with these variables. If you know a variable contains thread-specific information, make it local. You can make a variable global if you know the information in it will be used by other threads.Adhering to such practices can become even more convoluted when single threaded programs are converted to run as multi-threaded programs have threads being run in parallel. For example:

```
DWORD N=0;
....
for(int i=0; i < MAX; ++i) {
...Allocate and initialize pArgs data ...
N += ThreadFunc(pArgs);
}
...
DWORD WINAPI ThreadFunc (ARGS pArgs)

{
DWORD result;
...
result=...; return result;
}
```

The code above is for a single-threaded program. The program executes sequentially. The thread function is called repeatedly in a for loop. When a thread is called, the information it returns is stored in N, and the process is repeated for the next thread and so on. Because this is a single- threaded program, only one thread will be executed at a time. If you alter this code to a multi- threaded program, you will face issues.

Topic No – 137: (How to write thread Safe Code)

Guidelines for writing thread safe code to ensure that the code runs smoothly in a threaded environment. When more than one thread can run the same code without introducing synchronization issues, it is said to be thread safe.

1. Variables that are required locally should not be accessible globally. They should be placed on Stack or in Data Structure passed to thread or the thread TLS.
2. If a function is being used by several thread and it contains a variable that is thread specific such as a counter than store it in TLS or thread dedicated DS. Do not store it in global memory.
3. Avoid race conditions. If some required variables are uninitialized then create suspended threads until variables are initialized. If some condition needs to be met before a code block is executed, then make sure the condition is met by waiting on synchronization objects.
4. Thread should not change the process environment. Changing environment by one thread will affect other threads. Thread should not change standard input or output devices. Also, it should not change environment variables.
5. All variable that are meant to be shared among threads are either kept global or static. They are protected by synchronization or interlocked mechanism using memory barriers.

Topic No – 138: (Thread Synchronization Objects)

Windows support different types of synchronization objects. The objects can be used to enforce synchronization and mutual exclusion.

Synchronization Mechanism

One method that has been used is by **WaitForSingleObject()** or **WaitForMultipleObjects()**. Using this method, a thread waits for other threads to terminate. Other method used File Locks when multiple processes tries to access a file. Windows provides four different synchronization objects i.e., Mutexes, Semaphores, Events and Critical Section.

Risks in Using Synchronization Objects

There are always inherent risks involved in the use of such objects such as deadlocks. Care is required while using these objects. In higher version of Windows other objects like SRW locks and Condition variables are also available. Other advanced objects are waitable timers and IO completion ports.

Topic No – 139: (CRITICAL_SECTION Objects)

Critical Section

Critical section is the part of the code that can only be executed by one thread at a time. Windows provides the CRITICAL_SECTION objects as a simple lock mechanism for solving critical section problem. CRITICAL_SECTION Objects are initialized and deleted but do not have handles and are not shared among processes. Only one thread at a time can be in the CS variable, although many threads may enter and leave CS at numerous instances.

The function used to initialize and delete CRITICAL_SECTION are

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

When a thread wants to enter into the critical section it must call EnterCriticalSection() and when it leaves it should call LeaveCriticalSection().

A call to LeaveCriticalSection() must match an EnterCriticalSection(). Multiple threads may call EnterCriticalSection() but only one thread is allowed to enter the critical section while the rest are blocked. A blocked thread can enter the critical section when LeaveCriticalSection() is invoked.

Recursive Critical Section

If one thread has entered a CS it can enter again. Windows maintain a count. Thread will have to leave as many times as it enters. This is implemented to support recursive functions and to make shared library functions thread safe. There is no timeout to EnterCriticalSection(). A thread will remain blocked forever if a matching Leave call is not received. A thread will remain blocked forever if a matching Leave call is not received.

However, as thread can always poll to see whether another thread owns a CS using the function:

```
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

If the function returns **TRUE**, then it indicates that the current thread now owns the CS. If it returns false, then it indicates that the CS is owned by some other thread and it is not safe to enter CS.

Since CRITICAL_SECTION is a user space object therefore it has apparent

advantages over other kernel space objects.

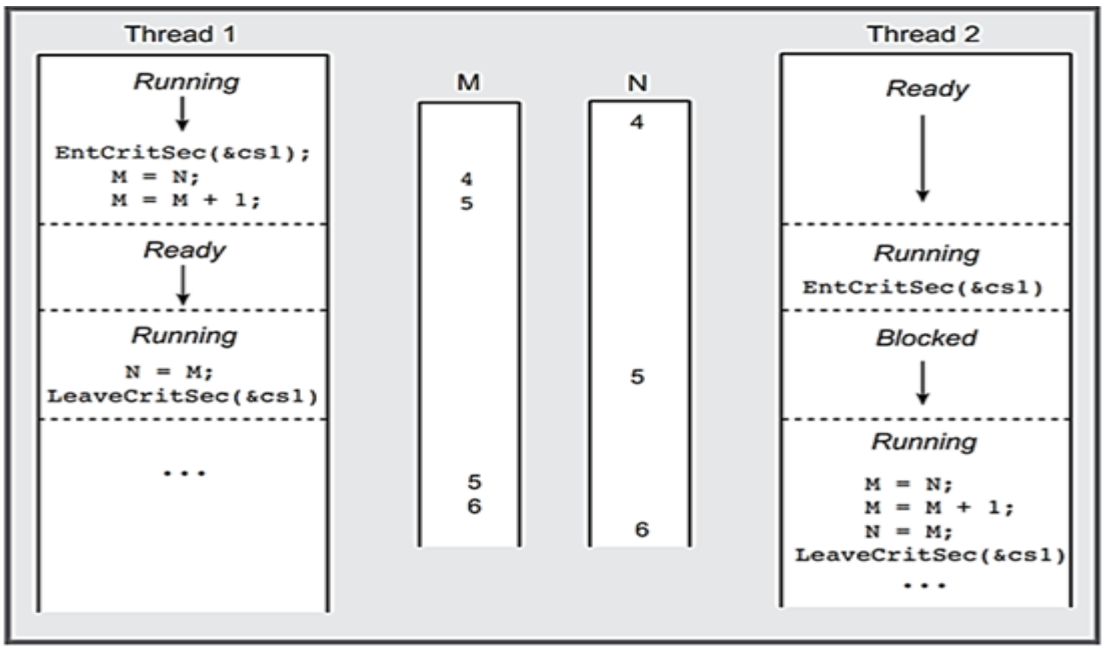
Topic No – 140: (CRITICAL_SECTION for Protecting Shared Variables)

In this module, we will study how to use protected shared variables in critical section and how critical section can assist us use protected shared variables.

Using the critical section construct is easy and intuitive. Consider an example of a server that maintains status related information in shared variables, information like number of requests, number of responses and requests currently under process. Since such count variables are shared therefore only one thread can be allowed to modify it at a time.

CRITICAL_SECTION construct can be used easily to ensure this. In this solution also an intermediate variable is used to emphasize the role of CRITICAL_SECTION.

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0;
/* N is a global variable, shared by all threads. */
InitializeCriticalSection (&cs1);
    . . .
/* Create one or more threads using ThreadFunc */
    . . .
DWORD WINAPI ThreadFunc (TH_ARGS pArgs);
{
    DWORD M;
    __try {
        EnterCriticalSection (&cs1);
    if (N < N_MAX) { M = N; M += 1; N = M; }
    } __finally {
        LeaveCriticalSection (&cs1)
    }
}
    ...
DeleteCriticalSection (&cs1);
```



Topic No 141: (Protect a Variable with a Single Synchronization Object)

In this module, we will discuss a guideline how to protect shared variables, how to use synchronization object, and what kind of mapping is there between synchronization object and variables like one to one or one to many.

Single Synchronization Object

All the variables within the critical section must be guarded by a single object. Using different objects within the same thread or using different objects across numerous threads sharing same data would be incorrect. Shared variables must be protected by a single object across all threads for mutual exclusion to work.

Below is given an example of incorrect use of synchronization object. In this example, two different objects are using the same variable N. Such code may generate incorrect results therefore share all variables using a single object.

```
DWORD WINAPI ThreadFunc (ARGS pArgs)
{
    . . .
    EnterCriticalSection(&cs1);
    N += 5;
    LeaveCriticalSection(&cs1);
    . . .
    InterlockedDecrement(&N);
    . . .
    EnterCriticalSection(&cs2);
    N -= 5;
    LeaveCriticalSection(&cs2);
    . . .
}
```

Week 13: Topic 142-153

Topic No 142: (Producer-Consumer Problem)

In this module, we will discuss a version of Producer-Consumer problem.

Producer-Consumer Threads

Producer consumer problem is a classical problem in mutual exclusion. It has many versions. Here we describe a simplistic version. This version clearly shows how to build and protect data structures for storing objects. Also, we discuss how to establish invariant properties of variables which are always TRUE outside CS.

- In addition to primary threads there are two more threads: a producer and a consumer thread.
- The producer periodically creates a message. The message is contained in a table.
- The consumer on request of the user displays the message. The displayed data must be most recent, and no data should be displayed twice.
- Do not display data while it is being updated by the producer. Do not display old data. Some of the produced messages may never be used and maybe lost. This is also like the pipeline model in which a message moves from one thread to another.
- The producer also computes a simple checksum of the message. Consumer makes sure by checking the checksum. If the consumer accesses the table while it is being updated the table will be invalid. CS ensures that this does not happen. The invariant is that the checksum is correct for current message contents.

Topic No 143: Sample Program for Producer-Consumer Problem Producer-Consumer

Producer consumer problem is a classical problem in mutual exclusion. Based on limitations and function described the following program is developed:

```
/* Chapter 9. simplePC.c */
/* Maintain two threads, a Producer and a Consumer */
/* The Producer periodically creates checksummed mData buffers, */
/* or "message block" which the Consumer displays when prompted */
/* by the user. The consumer reads the most recent complete */
/* set of mData and validates it before display */

#include "Everything.h" #include <time.h>
#define DATA_SIZE 256
typedef struct MSG_BLOCK_TAG { /* Message block */
        CRITICAL_SECTION mGuard;  /* Guard the message block structure */
        DWORD fReady, fStop;
                /* ready state flag, stop flag  */
        volatile DWORD nCons, mSequence; /* Message block mSequence number        */
        DWORD nLost;
time_t mTimestamp;
        DWORD mChecksum; /* Message contents mChecksum     */
        DWORD mData[DATA_SIZE]; /* Message Contents   */
} MSG_BLOCK;
/*      One of the following conditions holds for the message block         */
/*      1)      !fReady || fStop        */
/*      nothing is assured about the mData  OR      */
/*      2)      fReady && mData is valid      */
/*      && mChecksum and mTimestamp are valid  */
/* Also, at all times, 0 <= nLost + nCons <= mSequence                  */
/* Single message block, ready to fill with a new message    */
MSG_BLOCK mBlock = { 0, 0, 0, 0, 0 };
```

```
DWORD WINAPI Produce (void *);
DWORD WINAPI Consume (void *);
void MessageFill (MSG_BLOCK *);
void MessageDisplay (MSG_BLOCK *);
int _tmain (int argc, LPTSTR argv[])
{
        DWORD status;
        HANDLE hProduce, hConsume;
        /* Initialize the message block CRITICAL SECTION */
        InitializeCriticalSection (&mBlock.mGuard);
        /* Create the two threads */
        hProduce = (HANDLE)_beginthreadex (NULL, 0, Produce, NULL, 0, NULL);
        if (hProduce == NULL)
                ReportError (_T("Cannot create Producer thread"), 1, TRUE);
        hConsume = (HANDLE)_beginthreadex (NULL, 0, Consume, NULL, 0, NULL);
        if (hConsume == NULL)
                ReportError (_T("Cannot create Consumer thread"), 2, TRUE);
        /* Wait for the Producer and Consumer to complete */
        status = WaitForSingleObject (hConsume, INFINITE);
        if (status != WAIT_OBJECT_0)
                ReportError (_T("Failed waiting for Consumer thread"), 3, TRUE);
        status = WaitForSingleObject (hProduce, INFINITE);
        if (status != WAIT_OBJECT_0)
                ReportError (  T("Failed waiting for Producer thread"), 4, TRUE);
        DeleteCriticalSection (&mBlock.mGuard);
        _tprintf (_T("Producer and Consumer threads have terminated\n"));
        _tprintf (_T("Messages Produced: %d, Consumed: %d, Lost: %d.\n"),
        mBlock.mSequence, mBlock.nCons, mBlock.mSequence - mBlock.nCons);
return 0;
}
DWORD WINAPI Produce (void *arg)
```

```
/* Producer thread - Create new messages at random intervals */
{
        srand ((DWORD)time(NULL)); /* Seed the random # generator       */
        while (!mBlock.fStop) {
                /* Random Delay */
                 Sleep(rand()/100);
                /* Get the buffer, fill it */
                EnterCriticalSection (&mBlock.mGuard);
                        try {
                                if (!mBlock.fStop) {
                                        mBlock.fReady = 0;
                                         MessageFill (&mBlock);
                                         mBlock.fReady = 1;
                                        InterlockedIncrement (&mBlock.mSequence);
                                }
                        }
                         finally { LeaveCriticalSection (&mBlock.mGuard); }
                }
                return 0;
}
DWORD WINAPI Consume (void *arg)
{
CHAR command, extra;
/* Consume the NEXT message when prompted by the user */
while (!mBlock.fStop) { /* This is the only thread accessing stdin, stdout */
_tprintf (_T("\n**Enter 'c' for Consume; 's' to stop: "));
_tscanf (_T("%c%c"), &command, &extra); if (command == _T('s')) {
/* ES not needed here. This is not a read/modify/write.
* The Producer will see the new value after the Consumer returns */ mBlock.fStop = 1;
} else if (command == _T('c')) { /* Get a new buffer to Consume */ EnterCriticalSection
(&mBlock.mGuard);
 try {
```

```
if (mBlock.fReady == 0)
_tprintf (_T("No new messages. Try again later\n"));
else {
MessageDisplay (&mBlock);
mBlock.nLost = mBlock.mSequence - mBlock.nCons + 1; mBlock.fReady = 0; /* No new messages
are ready */ InterlockedIncrement(&mBlock.nCons);
}
}

  finally { LeaveCriticalSection (&mBlock.mGuard); }
} else {
_tprintf (_T("Illegal command. Try again.\n"));
}
}
return 0;
}
void MessageFill (MSG_BLOCK *msgBlock)
{
/* Fill the message buffer, and include mChecksum and mTimestamp */
/* This function is called from the Producer thread while it */
/* owns the message block mutex*/

DWORD i;
msgBlock->mChecksum = 0;
for (i = 0; i < DATA_SIZE; i++) {
msgBlock->mData[i] = rand();
msgBlock->mChecksum ^= msgBlock->mData[i];

}
msgBlock->mTimestamp = time(NULL); return;
}
```

```
void MessageDisplay (MSG_BLOCK *msgBlock)

{

/* Display message buffer, mTimestamp, and validate mChecksum */

/* This function is called from the Consumer thread while it           */

/* owns the message block mutex     */ DWORD i, tcheck = 0;


for (i = 0; i < DATA_SIZE; i++)

tcheck ^= msgBlock->mData[i];

_tprintf (_T("\nMessage number %d generated at: %s"),

msgBlock->mSequence, _tctime (&(msgBlock->mTimestamp)));

_tprintf (_T("First and last entries: %x %x\n"),

msgBlock->mData[0], msgBlock->mData[DATA_SIZE-1]); if (tcheck == msgBlock->mChecksum)
_tprintf (_T("GOOD ->mChecksum was validated.\n"));

else

_tprintf (_T("BAD ->mChecksum failed. message was corrupted\n"));


return;

}
```

Topic No 144: (Mutexes)

In this module we will discuss mutexes. Mutex, a short form of Mutual Exclusion. Windows provides an object called a mutex. Using this object, we can enforce mutual exclusion. Mutexes have some advantages beyond CRITICAL_SECTION. Mutexes can be named and have handles. They can also be used for interprocess communication between threads in different processes.

For example, if two processes share files through memory maps, then mutexes can be used for protection. Mutexes also allow timeout values. Mutexes can automatically become signaled once abandoned by the terminating thread. A thread gains ownership to mutex by waiting successfully on mutex handle using **WaitForSingleObject()** or **WaitForMultipleObject().**

Ownership is released using ReleaseMutex(). A thread should be careful about releasing a thread as soon as possible. A thread can acquire a single mutex several times. A thread will not be blocked if it already has ownership. The recursive property holds for Mutexes also.

Windows functions used to manage mutexes are

- CreateMutex()
- ReleaseMutex()
- OpenMutex()

```
HANDLE CreateMutex (
                        LPSECURITY_ATTRIBUTES lpsa,
                        BOOL bInitialOwner,
                        LPCTSTR lpMutexName );
```

1. **bInitialOwner** If it is TRUE, handover ownership to the calling thread immediately. However, it is ignored if the mutex already exists as per its name.
2. **lpMutexName** is the name of the mutex. The name is assigned as per rules of the Windows namespace.

   If NULL is returned it indicates failure.

OpenMutex () is used to open and existing named mutex. The Open operation is followed by Create. The main thread would usually create a mutex while other threads can open it. This construct allows to synchronize threads from different processes.

```
BOOL RealseMutex (HANDLE hMutex);
```

Similarly **ReleaseMutex**() releases the ownership of the Mutex for a thread. It fails if the Mutex is not already owned by the thread.

Topic No 145: (Mutexes, CRITICAL_SECTIONS and Deadlocks)

In this module we will discuss deadlocks. Deadlocks may arise when Mutexes or CRITICAL SECTIONS are used.

Using Concurrency Objects

Concurrency objects must be used carefully, otherwise it can lead to a deadlock situation. Deadlocks are a byproduct of concurrency control. It occurs when two or more objects try to lock a resource at the same time.

An Example

For instance, there are two lists (A and B) with the same structure maintained by different worker threads. In one situation, it can be possible that an operation is only allowed if a certain element is either present in both the lists or none. An operation is invalid if an element is present in just one.

In another situation, an element in one list cannot be in the other. Based on these situations, concurrency objects are required for both lists. Using a single mutex for both lists will degrade performance by restricting concurrent updates.

```
static struct {
   /* Invariant: List is a valid list. */
   HANDLE guard; /* Mutex handle. */
   struct ListStuff;
}  ListA, ListB;
...
DWORD WINAPI AddSharedElement (void *arg)
/* Add a shared element to lists A and B. */
{ /* Invariant: New element is in both or neither list. */
   WaitForSingleObject (ListA.guard, INFINITE);
   WaitForSingleObject (ListB.guard, INFINITE);
   /* Add the element to both lists ... */
   ReleaseMutex (ListB.guard);
   ReleaseMutex (ListA.guard);
   return 0;
}
DWORD WINAPI DeleteSharedElement (void *arg)
/* Delete a shared element to lists A and B. */
{
   WaitForSingleObject (ListB.guard, INFINITE);
   WaitForSingleObject (ListA.guard, INFINITE);
   /* Delete the element from both lists ... */
   ReleaseMutex (ListB.guard);
   ReleaseMutex (ListA.guard);
   return 0;
}
```

Avoiding deadlock situation

One method to avoid this situation is to maintain a hierarchy. All threads should follow the same hierarchy. They should acquire and release mutexes in the same order. In the example, the situation can be easily avoided if both the threads acquire mutexes in the same order.

Another technique that can be used is an array of mutexes and using **WaitForMultipleObjects()** with the **fWaitAll** flag. In this case, the thread will either acquire both the mutexes or none. This technique is not possible with CRITICAL_SECTION.

Topic No 146: (Mutexes and CS)

This module is about Mutexes VS CRITICAL_SECTIONs. Mutexes and CS are similar in the sense that they are used to achieve the same goal. Both can be owned by a single thread, and other threads trying to gain access to a resource shall be denied access until the resource is released. However, there are few advantages of using mutexes against some

performance drawbacks.

Advantages of Mutexes

- Mutexes that are abandoned due to the abrupt termination of a thread are automatically signaled so that waiting threads are not blocked permanently. However, abrupt termination of a thread indicates a serious programming flaw.
- Mutex waits can time out, whereas CS wait does not.
- As they are named, mutexes are shareable over numerous threads in different processes.
- Threads that create the mutexes can acquire immediate ownership (slight convenience).
- However, in most cases, CSs will work considerably faster.

Topic No 147: (Semaphores)

This module is about semaphores. Semaphore is simply a data structure that is used for concurrency control.

```
BOOL CreateSemaphore (
                    LPSECURITY_ATTRIBUTES lpsa,
                    LONG lSemInitial,
                    LONG lSemMax,
                    LPCTSTR lpSemName);
```

Semaphore Count Semaphore maintains a count. A semaphore is in a signaled state when the count is greater than 0. Semaphore is unsignaled when the count is 0. Threads use the wait function on semaphore. The count is decremented when a waiting thread is released. The following functions are used: **CreateSemaphore()**, **CreateSemaphoreEx(), OpenSemaphore**(), and **ReleaseSemaphore**().

1. **lSemMax** must be 1 or greater, which is the maximum value of semaphore count.
2. **lSemInitial** is the initial value.

   0<= lSemInitial<= lSemMax.

A NULL return in case of failure.

```
BOOL RealseSemaphore (
                        HANDLE hSemaphore,
                        LONG cReleaseCount,
                        LPLONG lpPreviousCount,
```

1. **lpPreviousCount** gives the count preceding the release.
2. **cReleaseCount** gives the count after the release and must be greater than 0. The call will fail and return FALSE if it would cause the count to exceed the maximum, and the count will remain unchanged. Also, in this case, the release count will not be valid.

Any thread can release the semaphore, not just the one that acquired its ownership. Hence, there is no concept of abandonment.

Topic No 148: (Using Semaphores)

This module is about using semaphores. We have already discussed semaphores. Now we will discuss how to use the structure and functions in a program to enforce mutual exclusion.

Semaphore Concepts

Classically, a semaphore count represents the number of available resources. Sem Maximum represents the number of resources. In a producer-consumer scenario, the producer would place an element in the queue and call **ReleaseSemaphore().** The consumer will wait on the semaphore and decrease the count after consuming the item.

Thread Creation

Previously, in many examples, numerous threads were created in a suspended state and were not activated until all the threads were created. This problem can be handled by semaphores without suspending threads. A newly created thread will wait on a semaphore with the semaphore initialized by 0. The boss thread will call **ReleaseSemaphore** with the count set to the number of threads.

Topic No 149: (Semaphore Limitation)

In this module we will discuss limitations of semaphore.

Limitations

Some limitations are encountered while using windows semaphore. How can a thread request to decrease the count by 2 or more? The thread will have to wait twice on the semaphore. Calling wait twice will not be atomic and execution may switch in between.

Below is given a code of two threads, thread1 on the left side and thread2 on the right side. The problem with the below code is that the wait functions are not atomic; they are separate wait functions. Because they are not single functions, switching between these two wait functions (after completion of first wait function) in thread1 may occur, and execution control may reach the thread2 wait function, resulting in a deadlock-like situation. So this is one of the limitations of semaphores.

```
/* hSem is a semaphore handle.
The maximum semaphore count is 2. */
   ...
/* Decrement the semaphore by 2. */
WaitForSingleObject (hSem, INFINITE);
WaitForSingleObject (hSem, INFINITE);
   ...
/* Release two semaphore counts. */
ReleaseSemaphore (hSem, 2, &PrevCount);
```

```
/* hSem is a semaphore handle.
The maximum semaphore count is 2. */
   ...
/* Decrement the semaphore by 2. */
WaitForSingleObject (hSem, INFINITE);
WaitForSingleObject (hSem, INFINITE);
   ...
/* Release two semaphore counts. */
ReleaseSemaphore (hSem, 2, &PrevCount);
```

How can we deal with this limitation? Make the two wait functions atomic by using a CriticalSection to avoid switching during the wait, as shown in the below code.

```
/* Decrement the semaphore by 2. */
EnterCriticalSection (&csSem);
WaitForSingleObject (hSem, INFINITE);
WaitForSingleObject (hSem, INFINITE);
LeaveCriticalSection (&csSem);
   ...
ReleaseSemaphore (hSem, 2, &PrevCount);
```

Other Solutions

Another solution one can suggest is the use of **WaitForMultipleObjects()** with an array holding multiple references to the same semaphore. However, this solution will fail readily as the call to **WaitForMultipleObjects()** fails when it detects duplicate objects in the array. Secondly, all the handles may get signals even if the count is 1.

Topic No 150: (Events)

In this module, we will discuss Events. Events are synchronization objects like Mutexes, Semaphores and Critical Section that can be used for concurrency control.

Events can indicate to other threads that a specific condition now holds, such as some message being available. Multiple threads can be released from wait simultaneously when an event is triggered. Events are classified as either manual-reset or auto-reset. Event property is set using **CreateEvent()**.

A manual-reset event can signal several waiting threads simultaneously and can be reset. An auto-reset event signals a single waiting thread, and the event is reset automatically. The functions used to handle events are **CreateEvent(), CreateEventEx(), OpenEvent(), SetEvent(), ResetEvent() and PulseEvent().**

```
HANDLE CreateEvent(
                    LPSECURITY_ATTRIBUTES lpsa,
                    BOOL bManualReset,
                    BOOL BInitialState,
                    LPTCSTR lpEvenName);
```

The function creates an event. **bManualReset** is set to TRUE to set a manual reset event. If **bInitialState** is TRUE, the event is set to a signaled state. A named event can be opened using **OpenEvent()** from any process.

```
BOOL SetEvent (HANDLE hEvent)
BOOL ResetEvent (HANDLE hEvent)
BOOL PulseEvent (HANDLE hEvent)
```

The above three functions are used to control events. A thread can signal the event by using **SetEvent()**. In the case of auto-reset, a single thread is released out of many. Event automatically returns to a non-signaled state. If no threads are waiting, the event remains in a signaled state until a thread waits on it. In this case, the thread will be immediately released.

If the event is manual-reset, it remains signaled until a thread explicitly calls **ResetEvent().** Meanwhile, all the waiting threads are released. Consequently, other threads may also wait to be released immediately before the reset.

PulseEvent() releases all threads currently waiting for a manual reset. The event is automatically reset. In the case of an auto-reset event, PulseEvent() releases a single waiting thread, if any.

Topic No 151: (Event Usage Models)

In this module, we will discuss Event Usage Models. Event can be implemented using different types of models. These models are designed according to the usage.

Combinations

There can be four distinct ways to use events. These four models culminate from combinations of SetEvent(), PulseEvent() and use of auto and manual events. Each combination proves useful depending on the situation. The combinations are highlighted in the table.

	Auto-Reset Event	Manual-Reset Event
`SetEvent`	Exactly one thread is released. If none is currently waiting on the event, the first thread to wait on it in the future will be released immediately. The event is automatically reset.	All currently waiting threads are released. The event remains signaled until reset by some thread.
`PulseEvent`	Exactly one thread is released, but only if a thread is currently waiting on the event. The event is then reset to nonsignaled.	All currently waiting threads, if any, are released, and the event is then reset to nonsignaled.

Understanding Events

An auto-reset event can be conceptualized as a door that automatically shuts when opened. Whereas the manual reset does not shut automatically when opened. Hence, **PulseEvent()** can be considered as a door that is open which shuts when a single thread passes through in the case of auto-reset. While in the case of a manual reset, multiple waiting threads can pass through. SetEvent(), on the other hand, simply opens the door and releases a thread.

Topic No 152: (Producer-Consumer Solution Using Events)

In this module, we will try to solve Producer-Consumer problem using Events.

Producer Consumer

Here is another example that provides a solution to our Producer-Consumer problem using Events. The solution uses Mutexes rather than CSs. A combination of auto-reset and SetEvent() is used in the producer to ensure only one thread is released. Mutexes in the program make access to the message data structure mutually exclusive while an event is used to signal the availability of new messages.

```
#include "Everything.h"
 #include <time.h>
#define DATA_SIZE 256
typedef struct MSG_BLOCK_TAG { /* Message block */
                HANDLE      mGuard;      /* Mutex to uard the message block structure*/
                HANDLE      mReady;      /* "Message ready" auto-reset event */
                DWORD       fReady, fStop; /* ready state flag, stop flag */
                volatile DWORD mSequence; /* Message block mSequence number*/
volatile DWORD nCons, nLost;
                time_t mTimestamp;
                DWORD       mChecksum;   /* Message contents mChecksum*/
                DWORD       mData[DATA_SIZE]; /* Message Contents*/
} MSG_BLOCK;
/*One of the following conditions holds for the message block */
/*1)     !fReady || fStop*/
/*nothing is assured about the mData          OR */
/*2)     fReady && mData is valid*/
/* && mChecksum and mTimestamp are valid*/
/* Also, at all times, 0 <= nLost + nCons <= mSequence*/
/* Single message block, ready to fill with a new message    */
MSG_BLOCK mBlock = { 0, 0, 0, 0, 0 };
DWORD WINAPI Produce (void *);
 DWORD WINAPI Consume (void *);
void MessageFill (MSG_BLOCK *);
void MessageDisplay (MSG_BLOCK *);
int _tmain (int argc, LPTSTR argv[])
          {
          DWORD status;
          HANDLE hProduce, hConsume;
          /* Initialize the message block mutex and event (auto-reset) */
           mBlock.mGuard = CreateMutex (NULL, FALSE, NULL);
           mBlock.mReady = CreateEvent (NULL, FALSE, FALSE, NULL);
```

```
                /* Create the two threads */
                hProduce = (HANDLE)_beginthreadex (NULL, 0, Produce, NULL, 0, NULL);
                if (hProduce == NULL)
                        ReportError (_T("Cannot create producer thread"), 1, TRUE);
                hConsume = (HANDLE)_beginthreadex (NULL, 0, Consume, NULL, 0, NULL);
                 if (hConsume == NULL)
                        ReportError (_T("Cannot create consumer thread"), 2, TRUE);
                /* Wait for the producer and consumer to complete */
                status = WaitForSingleObject (hConsume, INFINITE);
                if (status != WAIT_OBJECT_0)
                        ReportError (_T("Failed waiting for consumer thread"), 3, TRUE);
        status = WaitForSingleObject (hProduce, INFINITE);
        if (status != WAIT_OBJECT_0)
                ReportError (  T("Failed waiting for producer thread"), 4, TRUE);
        CloseHandle (mBlock.mGuard);
        CloseHandle (mBlock.mReady);
        _tprintf (_T("Producer and consumer threads have terminated\n"));
_tprintf (_T("Messages produced: %d, Consumed: %d, Known Lost: %d\n"), mBlock.mSequence,
mBlock.nCons, mBlock.mSequence - mBlock.nCons);
        return 0;

}

DWORD WINAPI Produce (void *arg)
/* Producer thread - Create new messages at random intervals */
{
        srand ((DWORD)time(NULL));/* Seed the random # generator        */
while (!mBlock.fStop) {
                /* Random Delay */
                Sleep(rand()/5); /* wait a long period for the next message */
                /* Adjust the divisor to change message generation rate */
                /* Get the buffer, fill it */
```

```
                WaitForSingleObject (mBlock.mGuard, INFINITE);
                 try {
                        if (!mBlock.fStop) {
                                mBlock.fReady = 0; MessageFill (&mBlock);
                                 mBlock.fReady = 1;
InterlockedIncrement(&mBlock.mSequence);
                                SetEvent(mBlock.mReady); /* Signal that a message is ready. */
                        }
                }
                 finally { ReleaseMutex (mBlock.mGuard); }
                }
                return 0;
                }
DWORD WINAPI Consume (void *arg)
{
DWORD ShutDown = 0;
        CHAR command[10];
        /* Consume the NEXT message when prompted by the user */
        while (!ShutDown) { /* This is the only thread accessing stdin, stdout */
                _tprintf (_T ("\n**Enter 'c' for Consume; 's' to stop: "));
                _tscanf_s (_T("%9s"), command, sizeof(command)-1);
                if (command[0] == _T('s')) {
                        WaitForSingleObject (mBlock.mGuard, INFINITE);
                        ShutDown = mBlock.fStop = 1;
                        ReleaseMutex (mBlock.mGuard);
} else if (command[0] == _T('c')) { /* Get a new buffer to consume */
                        WaitForSingleObject (mBlock.mReady, INFINITE);
                        WaitForSingleObject (mBlock.mGuard, INFINITE);
                         try {
                                if (!mBlock.fReady) _leave; /* Don't process a message twice */
                                /* Wait for the event indicating a message is ready
                                MessageDisplay (&mBlock);
```

```
                                InterlockedIncrement(&mBlock.nCons);
                                mBlock.nLost = mBlock.mSequence - mBlock.nCons;
mBlock.fReady = 0; /* No new messages are ready */
                        }
                         finally { ReleaseMutex (mBlock.mGuard); }
                } else {
                        _tprintf (_T("Illegal command. Try again.\n"));
                }
        }
        return 0;
                }
void MessageFill (MSG_BLOCK *msgBlock)
{
/* Fill the message buffer, and include mChecksum and mTimestamp        */
        /* This function is called from the producer thread while it  */
        /* owns the message block mutex*/
        DWORD i;
        msgBlock->mChecksum = 0;
        for (i = 0; i < DATA_SIZE; i++) {
                msgBlock->mData[i] = rand();
                msgBlock->mChecksum ^= msgBlock->mData[i];
}
        msgBlock->mTimestamp = time(NULL);
        return;
}
void MessageDisplay (MSG_BLOCK *msgBlock)
{
        /* Display message buffer, mTimestamp, and validate mChecksum */
        /* This function is called from the consumer thread while it */
        /* owns the message block mutex     */
        DWORD i, tcheck = 0;
        TCHAR timeValue[26];
```

```
for (i = 0; i < DATA_SIZE; i++)
        tcheck ^= msgBlock->mData[i];
    _tctime_s (timeValue, 26, &(msgBlock->mTimestamp));
    _tprintf (_T("\nMessage number %d generated at: %s"),
        msgBlock->mSequence, timeValue);
    _tprintf (_T("First and last entries: %x %x\n"),
        msgBlock->mData[0], msgBlock->mData[DATA_SIZE-1]);
    if (tcheck == msgBlock->mChecksum)
        _tprintf (_T("GOOD ->mChecksum was validated.\n"));
    else
        _tprintf (_T("BAD ->mChecksum failed. message was corrupted\n"));
    return;
}
```

Topic 153: Windows Synchronization Objects

Windows operating system is a multithreaded kernel that provides support for real time applications and multiprocessors. Program design and performance can be simplified and improved by threads, but it requires much care to use threads in case of shared resources at the same time. Process Synchronization is a way to coordinate processes that use shared data To synchronize access to a resource we use the following synchronization objects.

a) Critical Section
b) Semaphore
c) Mutex
d) Event

Week 14: Topic 154-165

Topic No -154: (Programming Guideline using Mutexes and CSs)

To minimize the concurrency related issues and errors following are the main guidelines.

1. If there is no time-out associated with ***WaitForSingleObject*** on a mutexhandle (CSs do not have a time-out), the calling thread could block forever, so a programmer should take care of it.
2. When a thread is in the critical section (CS) and it terminates without releasing the critical section then it will be impossible for the other thread in the queue to complete its execution and be blocked.
3. Mutexes have a way out for the above problem in critical section, if a thread or process owns a mutex and it terminates, then mutex has an abandonment property, abandoned mutex handles are signaled, it is a useful feature not available with CSs. It abandons the process and other threads are not blocked and fail as in CS case. Mutex abandonment of a process shows a flaw in the programming.
4. If you used the construct of **WaitForSingleObject()** and also specified the timeout then you must program in such a way that it checks if the wait ends due to timeout then it must release the critical resource.
5. Exactly one of the waiting threads at a time should be given the ownership of mutex. Only the OS Scheduler decides which thread has the priority according to its scheduling policy. Program should not assume the priority of any particular thread over the other.
6. A single mutex can be used to define several critical regions for several threads which are competing for the resource.
7. To ensure optimization critical Region size should be minimum. Programmers must ensure that critical regions cannot be used concurrently.
8. Performance decrease because of locks so minimize the use of locks inside the critical region.
9. The same data structure that stores the resource should also be used to store the mutexes because mutexes correspond to the resources.
10. Invariant is the property that assures whether you have enforced the concurrency correctly or not.
11. Complex conditions and decision structures should be avoided for entering into the critical region. Each critical region must have one entry and one exit.

12. Must ensure that mutex is locked on entry and unlocked on exit.
13. Avoid premature exits from the critical region such as break, return or goto statements, termination handlers are useful for protecting against such problems.
14. If the critical code region becomes too lengthy (longer than one page, perhaps), but all the logic is required, consider putting the code in a function so that the synchronization logic will be easy to read and comprehend.

Topic No -155: (More on interlocked functions)

The interlocked functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. They also perform operations on variables in an atomic manner. The threads of different processes can use these functions if the variable is in shared memory.

The **InterlockedIncrement** and **InterlockedDecrement** functions combine the steps involved in incrementing or decrementing a variable into an atomic operation. Interlocked functions are as useful as they are efficient; they are implemented using atomic machine instructions (for this reason, they are sometimes called “compiler intrinsic statements”).

Some Interlocked Functions and their details:

InterlockedExchange stores one variable into another.

```
Long InterlockedExchange(
                        LONG volatile *Target,
                        LONG Value);
```

It **returns** the previous value of *Target and sets its value to Value.

```
Long InterlockedExchangeAdd(
                        LONG volatile *Addend,
                        LONG Increment);
```

Increment is added to the *addend and the original value of *addend is returned.

Topic No -156: (More on interlocked functions)

When numbers of threads are running concurrently, there is a competition for resources, memory is also a resource and threads compete for the memory. So, we have to devise a way to reduce or optimize this competition to improve the performance. When we allocate a memory dynamically it is allocated from the heap. Threads allocate memory and free memory using **malloc ()** and **free ()** functions respectively.

Managing Access to Heap

a) Each thread that performs memory management can create a Handle to its own heap using **HeapCreate().** Memory allocation is then performed using using **HeapAlloc()** and **HeapFree()** rather than using **malloc()** and **free()**

b) Consider third party APIs for concurrent memory management.

Topic 157: Synchronization Performance Impact

Synchronization can impact on the performance following are the main reasons.

a) Locking, waiting, and even interlocked operations are time consuming.

b) Locking requires kernel operation and waiting is expensive.

c) Only one thread at a time can execute a critical code region, reducing concurrency and produces almost serialization execution.

d) Many processes when competing for memory and cache can produce unexpected effects.

Topic 158: (Gauging Performance Impact of Synchronization)

To enforce synchronization, we have many constructs like Critical Section, Mutexes, Semaphore. When we implement synchronization with these constructs there are some trade-offs performances wise.

We arrange data in the memory in such a way to avoid cache conflicts and allocation are aligned over optimal cache boundaries.

Following are the main modifiers used for minimal memory contention.

declpec(align), with this modifier memory is aligned in such a way to minimize contention.

Similarly **align_malloc()** and **align_free()** functions are used for

allocation/deallocation.

Example 9-1 (see in the book)

We developed different programs in different seniors.

1. Without using synchronization – NS
2. Using Critical Section -CS
3. Using Mutexes-MX
4. Using Interlocked Function-IN

Following are the main inferences we drew by running the programs using these scenarios. We basically compare the performance on the basis of following three aspects.

a) Real Time b)-User Time

b) System Time

Inferences

1. NS (no synchronization) and IN (interlocked functions) exhibit all most the same time for this example.
2. CS (Critical Section) version costs 2 times more than IN
3. MX(Mutexes) version costs more than 2 to 30 times than IN
4. Any type of locking (even IN) is more expensive than no locking, but we cannot discard it because of the significance of synchronization.

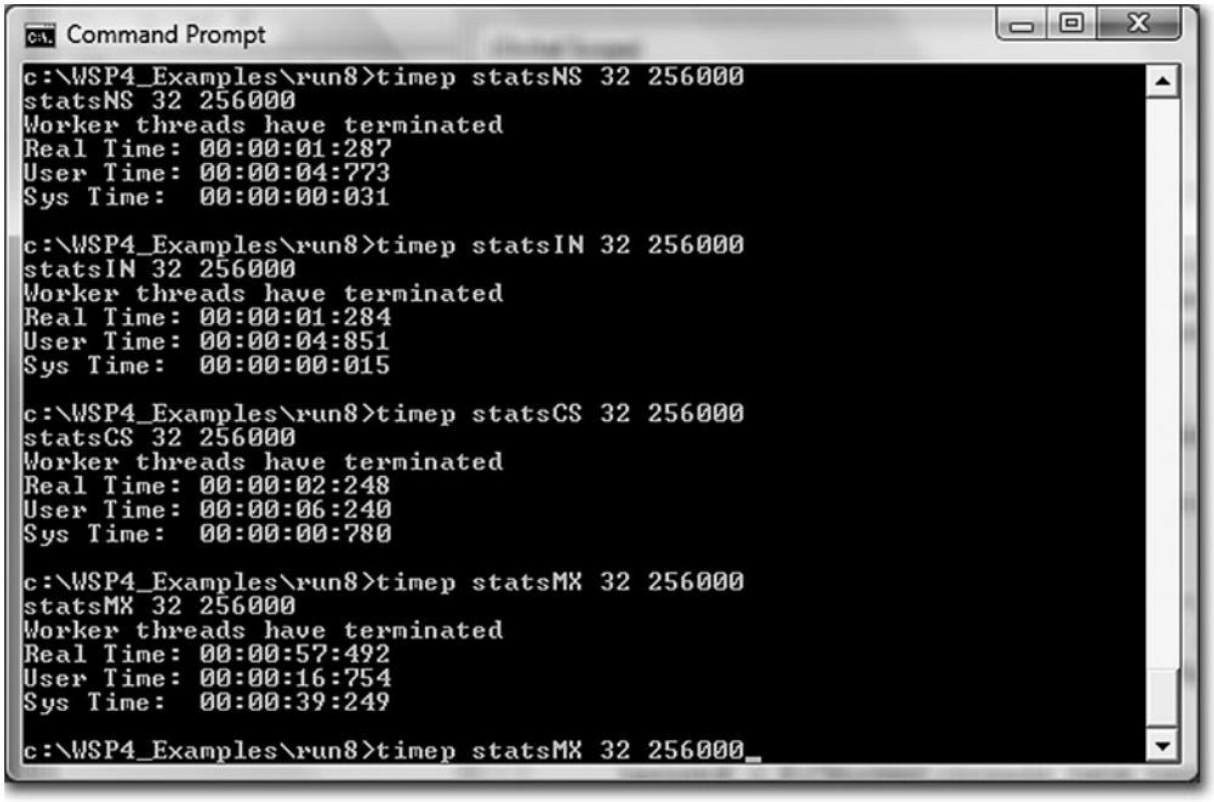
#Processes Clock Rate	StatsMX	statsCS
1, 1.4GHz	55.03	14.15
2, 2.0GHz	93.11	5.30
4, 2.4GHz	118.3	4.34
8, 1.7GHz	262.2	20.2

5. Mutexes are very slow, and unlike the behavior with CSs, performance degrades rapidly as the processor count increases. For instance, Table 9–1 shows the elapsed and times (seconds) for 64 threads and 256,000 work units on 1-, 2-, 4-, and 8-processor systems. CS performance, however, improves with processor count and clock rate.

Topic 159: (Performance Analysis of NS, IN, CS, and MX)

Gauging Performance:

Previously different programs were developed that performed a simple task using different concurrency control objects. Also, previously a program was developed that reported the real time, user time and system time a process takes to run. Now we use this program to analyze the running time of our programs.



Inferences:

- NS (no Synchronization) and IN (interlocked functions) exhibit almost the same time for this example.
- CS (critical section) version costs 2 times more than IN.
- MX (Mutex) version can cost more than 2 to 30 times IN.
- For older version of windows CS was not scalable. For recent versions its very scalable.
- Any type of locking (even IN) is more expensive than no locking. However, synchronization control has logical significance which cannot be discarded.
- Mutexes are slow and performance degrades as number of processors increase.
- The Table shows the performance of mutexes as the number of processors increase.

Topic 160: (False Sharing Contention)

While working with synchronization we face the problem of false sharing contention Cache has cache lines having different capacities, if you have a large array with different elements on a single line and if we access any element on that line then the whole line will be locked.

If on one line, we store more than one element then we have to face the consequences of false sharing contention. If any one element is locked on a single line, then other elements will also be locked and can only be accessed serially. False sharing contention occurs when two or more threads on different processes place variables on the same address line and a change in one variable will lock the whole cache line.

Solution

It will be better to align each variable in such a way that each variable is placed on a different cache line. We have to use this type of alignment in order to avoid false sharing. Of course, when we use this type of alignment it will be expensive memory wise.

Topic 161: (Tuning Performance with CS Spin Counts)

Different synchronization objects have different overheads. Critical Section overheads are moderate, but we can future optimize these overheads by fine tuning the spin counts. **Critical Section works in user space; it does not work in kernel space** and does not require complicated and/or convoluted system calls while mutexes invoke kernel calls for locking and unlocking like **ReleaseMutex()** requires system calls.

Lock bit is on

If any thread invoke CS with the help of **EnterCriticalSection()** it test its lock bit, if lock bit is off ,it mean no other thread is entered in the CS yet, so lock bit atomically set and operation proceed without waiting hence locking and unlocking of CS is very efficient and it just require a couple of machine level instructions.

The ID of the thread is stored in the CS data structure and it also stores the status of the recursive calls, so recursion overheads are also performed in the critical section.

Lock bit is off

If CS is already locked then after invoking **EnterCriticalSection()**, the thread will enter into the tight loop, thread calls tight loop because it involves all the processes of the multiprocessor system and tight loop will repetitively check the lock bit whether it is off or on.

Spin count will tell how much time the tight loop repetitively checks the **lock bit**.

If the lock bit repetitively checked without yielding the processor then it gives up calling **WaitForSingleObject()** -Spin count determines the number of times the loop repeats and it is used in multiprocessor system **LeaveCriticalSection()** turns off the lock bit and also informs the kernel by **ReleaseSemphore()** call in case there are any waiting threads.

Topic 162: (Setting the Spin Counts)

Spin count is important due to the fact that you can optimize the Critical Section with the help of spin count. How to set the spin count it will depend on many factors We can specify spin count when we initialize the Critical Section using the function **InitializeCriticalSectionAndSpinCount()**. The main advantage of optimizing the spin count is to avoid the system calls, when there are less system calls then overheads will be reduced.

We can set spin count initially and as well as dynamically on runtime using the function **SetCriticalSectionSpinCount()** According to MSDN 4000 is the good spin count for heap related functions but it depends, like if you have a short critical section then small spincount will work optimally. Spin count should be set as per the dynamics of the application and the number of processes.

Topic 163: (Slim Reader Writer Locks)

Slim reader writer locks are used to optimize the Synchronization and minimizing the overheads due to synchronization and mutual exclusion. Lightweight Reader Writer Locks has two modes.

a) **Exclusive Mode**

b) **Shared Mode**

In Exclusive mode they can be locked and writable -In shared mode reading is possible and different threads can read the data at a time in shared mode. SRWs can be used either in Exclusive Mode or in Shared mode, but you cannot upgrade or downgrade the Exclusive Mode or Shared Mode. so Exclusive mode cannot be converted into Shared mode and Shared mode cannot be converted to Exclusive mode once acquired. Threads have to decide the mode before using SRWs whether it is exclusive or shared.

SRWs are light weight and slim and the size of the associated pointers are 32 bit or 64 bits only. No Kernel objects are associated with SRWs and hence SRWs locks require

minimal resources.

SRWs do not support recursion unlike Critical Section, so SRW is simpler and faster.

The spin count value of SRW is preset optimally and cannot be set manually It is not imperative to delete the SRW.

Nonblocking calls are not associated with SRW.

Topic 164: (APIs for SRWs)

We use APIs to manage SRWs, Slim Read Write Constructs are light weight. Following are the SRW APIs

In order to Initialize the SRWs Lock we use

InitializeSRWLock()

Similarly, there are two APIs which can access SRWs in shared mode.

AcquiredSRWLockShared() and ReleaseSRWLockShared().

There are also two APIs which are used to access SRWs in Exclusive mode.
-AcquireSRWLockExclusive

-ReleaseSRWLockExclusive

Using SRWs

If any thread needs to read the shared data then we get the SRW in shared mode but the thread to write that data will use SRW in exclusive mode.

SRWs are used just like CS OR MUTEX in exclusive mode and are used in shared mode if the guarded variable is not changed by thread.

Topic 165: (Improved Locking through SRWs)

The main advantage of the SRWs is the shared mode, in shared mode you can read the locked variable. The thread which reads the locked variable need not to wait for the other threads to release it.

- Similarly, SRWs are light weight in implementation and resource requirement.
- The don't allow recursion because in recursion you have to maintain the stack and it will require time and more execution.

Evaluation SRWs performance by using threads shows that these are almost twice fast than critical Section.

Week 15: Topic 166-178

Topic 166: (Reducing Thread Contention)

Thread Limitations:

Just like memory contention systems also have to face the thread contention.As the number of threads increases, it causes serious performance issues for following reasons.

1. When a thread is created almost 1MB space is reserved for that thread and with increasing threads memory area is piled up.
2. With large number of threads, context-switching can become time consuming.
3. Thread context-switching can cause page faults.
4. If you minimize the threads, you can lose the benefits of parallelism and synchronization.

Optimizations:

Following are the main optimization techniques.

1. Use of semaphore throttles.
2. Using I/O completion ports
3. Asynchronous I/O
4. Using Thread Pools through callback functions
5. Asynchronous Procedure Calls (APC)
6. Third party frameworks like OpenMP and cilk++

Topic 167: (Semaphore Throttles)

The Solution:

- The solution is simple and can be incorporated into existing programs.
- The boss thread creates a semaphore with a small count say 4, depending upon the number of processors and other conditions.
- Each worker thread that wants to enter the critical section will wait on the semaphore before entering its critical section. The thread may use CS or Mutex to acquire the resource.
- The worker thread will release the semaphore after exiting its critical section.
- Overall, the CS or mutex contention will be minimized. Thread execution will be serialized among few threads waiting on mutex or CS.
- The boss thread can control the count dynamically depending upon the overall progress.
- However, a maximum count for semaphore is set initially.

```
WaitForSingleObject (hThrottleSem, INFINITE);
while (TRUE) { /* Worker loop */
   WaitForSingleObject (hMutex, INFINITE);
      ... Critical code section ...
   ReleaseMutex (hMutex);
} /* End of worker loop */
ReleaseSemaphore (hThrottleSem, 1, NULL);
```

More variations:

- If a specific worker thread is considered to be expensive then it can be made to wait longer using some programming technique.
- This technique works well with older version of windows. NT6 onwards have optimization techniques of their own. This technique should be used with care in case of NT6.
- This technique can also be applied to reduce contention in other resources like files, memory etc.

Topic 168: (Thread Pools)

- Thread pools are easy to use. Also already developed programs that uses threads can be easily modified to use thread pools.
- Application creates a work object rather than a thread. Each work object is submitted to the thread pool.
- Each work object has a call back function and is identified by a handle like structure.
- Thread pool manages a small number of "worker threads."
- Windows assigns worker threads to each worker object.
- Worker thread executes a worker object by invoking the callback function.
- When a worker object completes it returns to windows worker thread which in turn invokes another worker object.
- A work object should never invoke _endthreadex() within the callback function.
- Windows dynamically adjusts the number of worker threads based on process behaviour.
- The application can set upper and lower bounds for worker threads.
- There is a default thread pool. Also, the application can create additional pools.

Topic 169: (Thread Pools APIs)

- Thread pools are created and assigned thread objects
- The thread objects execute via a call back function
- Now, we discuss all the APIs for this purpose.

```
PTP_WORK CreateThreadpoolWork (
                                    PTP_WORK_CALLBACK pfnwk,
                                    PVOID pv,
                                    PTP_CALLBACK_ENVIRON pcbe);
```

- pfnwk is the pointer to the callback function
- pv is optional. It is a value that is passed to every call to the callback function.
- pcbe is the TP_CALLBACK_ENVIRON structure. Usually, default will sufficent.
- Return value is the thread pool work object. It is NULL in case of failure.

```
VOID SubmitThreadpoolWork (
                                    PTP_WORK pwk)
```

- This API is used to register a work object (i.e. A call back function and a parameter)
- The Windows Thread pool decides which and when to invoke the callback function of a work object
- pwk is the reference returned by the **CreateThreadpoolWork ()** call
- The callback function associated with pwk will be executed once. The thread used to run the callback function is determined by the kernel scheduler.
- Programmer do not need to manage threads, but synchronization still needs to be enforced.

```
VOID WaitForThreadpoolWorkCallbacks (
                                    PTP_WORK pwk,
                                    BOOL fCancelPendingCallBacks)
```

- The wait function does not have a timeout. It returns when all the callback functions

have returned.

- Optionally it can also cancel the work objects whose call back function has not started as yet using the second parameter. Callbacks that have started will run to completion.
- pwk is the work thread pool work object.

```
VOID CALLBACK WorkCallback (
                        PTP_CALLBACK_INSTANCE Instance,
                        PVOID Context,
                        PTP_WORK Work)
```

- Work is the work object and Context is pv value obtained from the work object creation call.
- The Instance is the callback instance that provide important information regarding the instance to enable kernel to schedule callbacks.
- CallbackMayRunLong informs the kernel that the callback instance may run for a longer period. Normally callback instances are expected to run for a short.

Topic 170: (Using Thread Pools)

- Callbacks are an efficient alternate to threading.
- Basic programming techniques in windows programming also use callback function primitively to produce a threading like effect.
- Here we use a previous threading example and modify it to use windows thread pool framework.

```
#include "Everything.h"
#define DELAY_COUNT 20
#define CACHE_LINE_SIZE 64
/* Usage: statsSRW_VTP nThread ntasks [delay [trace]]    */
/* start up nThread worker threads, each assigned to perform     */
/* "ntasks" work units. Each thread reports its progress    */
/* in its own unshard slot in a work performed array        */

VOID CALLBACK Worker (PTP_CALLBACK_INSTANCE, PVOID, PTP_WORK);
int workerDelay = DELAY_COUNT;
BOOL traceFlag = FALSE;
declspec(align(CACHE_LINE_SIZE))
typedef struct _THARG {
        SRWLOCK slimRWL;
        int objectNumber;
unsigned int tasksToComplete;
unsigned int tasksComplete;
} THARG;
int _tmain (int argc, LPTSTR argv[])
{
        INT nThread, iThread;
                HANDLE *pWorkObjects;
                SRWLOCK srwl;
        unsigned int tasksPerThread, totalTasksComplete;
        THARG ** pWorkObjArgsArray, *pThreadArg;
                TP_CALLBACK_ENVIRON cbe; // Callback environment
```

```
        if (argc < 3) {
                _tprintf (_T("Usage: statsSRW_VTP nThread ntasks [trace]\n"));
                return 1;
                }
_tprintf (_T("statsSRW_VTP %s %s %s\n"), argv[1], argv[2], argc>=4 ? argv[3] : "");
nThread = _ttoi(argv[1]); tasksPerThread = _ttoi(argv[2]);
if (argc >= 4) workerDelay = _ttoi(argv[3]);
traceFlag = (argc >= 5);
/* Initialize the SRW lock */
InitializeSRWLock (&srwl);
pWorkObjects = malloc (nThread * sizeof(PTP_WORK));
if (pWorkObjects != NULL)
pWorkObjArgsArray = malloc (nThread * sizeof(THARG *));
if (pWorkObjects == NULL || pWorkObjArgsArray == NULL)
ReportError (_T("Cannot allocate working memory for worke item or argument array."), 2,
TRUE); InitializeThreadpoolEnvironment (&cbe);
for (iThread = 0; iThread < nThread; iThread++) {
/* Fill in the thread arg */
pThreadArg = (pWorkObjArgsArray[iThread] = _aligned_malloc (sizeof(THARG),
CACHE_LINE_SIZE));
if (NULL == pThreadArg)
        ReportError (_T("Cannot allocate memory for a thread argument structure."),
3, TRUE);
pThreadArg->objectNumber = iThread; pThreadArg->tasksToComplete = tasksPerThread;
pThreadArg->tasksComplete = 0;
pThreadArg->slimRWL = srwl;
pWorkObjects[iThread] = CreateThreadpoolWork (Worker, pThreadArg, &cbe);
if (pWorkObjects[iThread] == NULL)
        ReportError (_T("Cannot create consumer thread"), 4, TRUE);
SubmitThreadpoolWork (pWorkObjects[iThread]);
}
/* Wait for the threads to complete */
```

```
for (iThread = 0; iThread < nThread; iThread++) {
/* Wait for the thread pool work item to complete */ WaitForThreadpoolWorkCallbacks
(pWorkObjects[iThread], FALSE); CloseThreadpoolWork(pWorkObjects[iThread]);
}
free (pWorkObjects);
_tprintf (_T("Worker threads have terminated\n"));
 totalTasksComplete = 0;
for (iThread = 0; iThread < nThread; iThread++) {
pThreadArg = pWorkObjArgsArray[iThread];
if (traceFlag) _tprintf (_T("Tasks completed by thread %5d: %6d\n"), iThread, pThreadArg-
>tasksComplete);
totalTasksComplete += pThreadArg->tasksComplete;
_aligned_free (pThreadArg);
}
free (pWorkObjArgsArray);
if (traceFlag) _tprintf (_T("Total work performed: %d.\n"), totalTasksComplete);
return 0;
}
VOID CALLBACK Worker (PTP_CALLBACK_INSTANCE Instance, PVOID Context, PTP_WORK
Work)
{
THARG * threadArgs; threadArgs = (THARG *)Context;
if (traceFlag)
_tprintf (_T("Worker: %d. Thread Number: %d.\n"), threadArgs->objectNumber,
GetCurrentThreadId());
while (threadArgs->tasksComplete < threadArgs->tasksToComplete) { delay_cpu (workerDelay);
AcquireSRWLockExclusive (&(threadArgs->slimRWL)); (threadArgs->tasksComplete)++;
ReleaseSRWLockExclusive (&(threadArgs->slimRWL));
}
return;
}
```

Topic 171: (Alternate Methods for Submitting Callbacks)

- In case the threads are simple an alternate and simplistic technique can be adopted.
- In this method work items can be omitted if they are not required.
- It uses the following APIs

```
BOOL TrySubmitThreadPoolCallback (
                                    PTP_SIMPLE_CALLBACK pfnwk,
                                    PVOID pv,
                                    PTP_CALLBACK_ENVIRON pcbe);
```

In this case the call uses a different type fpr callback function PTP_SIMPLE_CALLBACK

```
PTP_Work CreateThreadpoolWork (
                                    PTP_WORK_CALLBACK pfnwk,
                                    PVOID pv,
                                    PTP_CALLBACK_ENVIRON pcbe);
```

```
VOID CALLBACK SimpleCallback (
                                    PTP_CALLBACK_INSTANCE Instance,
                                    PVOID Context);
```

The callback function is defined in the above given form. The work items are excluded which are not required.

Topic No – 172: (The Process Thread pool)

According to MSDN, a thread pool is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is used to reduce the number of application threads and provide management of the worker threads.

The application that primarily performs parallel processing, or processes independent work items in the background, or performs an exclusive wait on Kernel objects can benefit from the **Threadpool**.

Every process has a dedicated thread pool. When a callback function is registered for a **threadpool**, then these functions are executed by the process thread pool. We cannot be

sure exactly how many threads are there in a specific process thread pool. The purpose of a thread is to take (or map) a callback function of a work object and start executing it. When a callback function finish executing then the corresponding thread becomes free and available for the Windows scheduler to be mapped with another work object.

If we have a lot of work objects but limited threads within the thread pool, then those work objects begin to compete with each other for having control of threads. This is normal behavior. It is the responsibility of the Windows kernel to resolve the competition. A system programmer has little control over this. If the programmer thinks that default threads in a thread pool are not sufficient, he/she may create more threads using **CreateThreadPool()** but it might provide benefits or it might further degrade the performance. depending on the situation.

Topic No - 173: (Other Thread Pool callback Types)

Whenever we create a work object, we also associate a callback function with it. **CreateThreadPoolWork()** associates a work object to a callback function and that callback function maps to a certain thread from the thread pool. A callback function typically performs computations and I/O.

There can be several types of callback functions. Some of them are listed below:

- **CreateThreadpoolTimer()** specifies a callback function to execute after a time interval and possibly periodically after that specified in milliseconds.
- **CreateThreadpoolIO()** allows to specify a callback function to execute when an overlapped I/O operation completes on a HANDLE.
- **CreateThreadpoolWait()** will allow you to register a callback function to execute when an object specified by Specified by **SetThreadpoolWait()** is signaled.

Topic No – 174: (Locking Performance)

We have discussed four locking mechanisms along with thread pools. Also, we compared the performance of each using different programs.

Many SW and HW factors vary the performance of each mechanism. However, we can generalize the performance in the following order from highest to lowest performance:

- **No Synchronization:** If we don't use any synchronization variable or locking mechanism, then obviously that code will runway faster as compared to using any synchronization construct.

But this is not a solution because we not only require faster execution but also accuracy. So, we have to implement some sort of synchronization construct.

- **Interlocked functions:** These are the fastest locking mechanisms.
- **SRW lock:** These locks work best if they are used with worker threads. They perform best when we use a mix of exclusive and shared locks.
- **SRW locks with conventional thread management:** If we do not use thread pool but use conventional thread management then performance marginally degrades.
- **Critical sections:** They don't perform well esp. when we have a lot of threads in our program. Their performance can be optimized by using spin counts.
- **Mutexes.** They have the worst performance. It can be marginally optimized using semaphore throttling.

Topic No – 175: (Extended Parallelism)

One of the first processors made by Intel had a clock speed of 477 MHz. As the time progresses, clock speed increases which means more instructions per second can be executed and hence increase in performance. But there is a bottleneck in clock speed - we cannot incrementally increase clock speed.

Most systems today, whether laptops or servers, will have clock rates in the 2–3GHz range. So the question arises how we can further enhance the computational performance given the clock speed bottleneck problem? Here comes the concept of multi-core processors - a single chip with multiple computational units or cores. The chip makers are marketing multicore chips with 2, 4, or more processors on a single chip. In turn, system vendors are installing multiple multicore chips in their systems so that systems with 4, 8, 16, or more total processors are common.

In the conventional programming paradigm, a programmer doesn't usually write programs keeping in mind the multi-core architecture. We have to implement parallel programming constructs like threading in our programs which is way more complex and different as compared to conventional serial.

programming. Therefore, if you want to increase your application's performance, you will need to

exploit the processor's inherent parallelism using threads. For example, the Boss/Worker model that we have discussed previously.

But it comes with a cost - we have to use synchronization constructs, concurrency control, mutual exclusion etc. When we frequently use these constructs, the performance might degrade. Also, if you increase and further increase computational cores in microprocessor chips, it doesn't guarantee that performance will also enhance. After a certain limit, the performance begins to drop.

Usage of Parallelism:

- Parallel sort and merge can be far more efficient.
- Fine grain parallelism should be exploited in divide and conquer based applications using recursion.
- Computational tasks that can be performed parallelly such as matrix multiplication, Fourier transformation, searching etc.
- Games and Simulations can also be decomposed in several parallel components. Importance of Parallelism:
- Parallelism is increasingly important because of the bottleneck in increasing processor cores.
- Learning programming skills is necessary to use the potential of current hardware.
- Most of the applications that do not use parallelism but yet are computationally intensive fail to use the full potential of HW.

Topic No – 176: (Parallel Programming Alternatives)

The most immediate step for writing a program that uses parallelism is to identify the parallel components within the program. So, you must be proficient in writing parallel programs and then pinpoint which components can be executed in parallel.

Once you have identified parallel components, the next step is to implement those components in a parallel manner. You must use any synchronization constraints like mutual inclusion etc.

There are several methods to implement parallelism within a windows program:

Method 1: Do It yourself (DIY)

- In this "do it yourself" (DIY) approach, thread management and synchronization is managed by the programmer.
- DIY approach is useful and effective in smaller programs with simple parallel structures.
- DIY can become complex and error prone while using recursion.

Method 2: Thread Pool

- Thread pool enables advanced kernel scheduling and resource allocation methods to enhance performance.
- However, these APIs are only available in NT6 and above.

Method 3: Use any Parallelism Framework

- It is the most widely accepted approach.
- They extend the Programming Language to express parallelism

Topic No – 177: (Parallelism Frameworks)

Frameworks have extensions of programming languages for expressing parallelism. Certain constructs and APIs sets are available in a specific framework that can be used in our programs to implement parallelism.

There are two sorts of parallelism:

- **Loop parallelism:** In loop parallelism, every loop iteration can execute concurrently. For example: matrix multiplication. In loop level parallelism our task is to extract parallel tasks from the loops. These parallel tasks will then be assigned to individual processor cores and hence it will reduce computational time.
- **Fork-Join parallelism:** In this type of parallelism, a function call can run independently from the calling program, which eventually must wait for the called function to complete its task. In fork- join parallelism, the control flow divides (like the shape of the fork) into multiple flows that join later.

Framework Features:

- These extensions/constructs/APIs may be in the form of compiler directives or primitives requiring compiler front end. Mostly the supported languages are C, C++, C#, Java and Fortran
- Run-time libraries are provided for efficient scheduling, locking and other related tasks.
- Support of results reduction. Results from parallel tasks are combined into one. For example, word counts from individual files are summed up.
- Parallel code can be serialized for debugging purposes such that the outcome remains unchanged.
- These frameworks contain tools for race detection and measuring parallelism.

Open-source frameworks:

- OpenMP: This framework is open source, portable and scalable. Numerous compilers support OpenMP like Visual C++. It supports multi-platform shared-memory parallel programming. Complete documentation can be found at: https://www.openmp.org/
- Intel Thread Building Blocks (TBB): It is a flexible performance library that contains a set of APIs which can add parallelism to applications. More information can be found at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.rodjo
- Cilk++: It adds extension to ordinary serial programming to perform parallel programming. It supports C and C++. More information can be found at: https://cilk.mit.edu/

Topic No – 178: (Challenges in Parallelism Programming)

Use of threads is direct and straightforward, however there are numerous pitfalls. To overcome these pitfalls, we have discussed many techniques. These challenges may also manifest while using parallelism frameworks.

- Identifying independent subtasks is not simple. Especially when a problem is encountered when working with older codes designed for serialized systems.
- Too many subtasks can degrade performance
- Too much locking can degrade performance.
- Global variables cause problems. Global variables may contain a count updated iteratively. Using parallelism this may need to be taken care of in parallel.
- Subtle performance issues arise due to memory cache architecture and multicore chips.

Final Week 16: Topic 179-190

Topic No – 179: (Processor Affinity)

Up till now it has been assumed that each thread is free to use any processor of a multiprocessor system. The kernel makes scheduling decisions and allocates processors to each thread. This is natural and conventional and is almost always the best approach. However, it is possible to assign a thread to a specific processor by setting processor affinity. A processor affinity is the ability to direct or bind a specific thread, or process, to use a specified core. When implementing processor affinity, our goal is to schedule a certain thread or a process on a certain subset of CPU cores.

Processor Affinity can be used in a number of ways. You can dedicate a processor to a small set of threads and exclude other threads from that processor. However, Windows can still schedule its own threads on the processor.

Advantage/Usage of defining Processor Affinity:

1. One of the advantages of defining processor affinity is to minimize delay caused due to memory barriers (concurrency control constructs etc.) For example, you can assign a collection of threads to a processor-pair that shares L2 Cache. Processor affinity can effectively decrease cache issues.
2. If you want to test a certain processor core(s) then processor affinity may be used for **diagnostic purposes**.
3. Worker threads that contend for a single resource can be allocated to a single processor by setting up processor affinity.

Topic No - 180 - Processor Affinity Masks

Each process has its own process affinity mask and a system affinity mask. These masks are basically bit vectors.

- **System Affinity Mask:** indicates the processors configured on this system.
- **Process Affinity Mask:** indicates the processors that can be used by the process threads. Or in other words the process mask indicates on which processor (cores) this process can be run. By default, it is the same as the system mask.
- **Thread Affinity Mask:** Each thread has a thread affinity mask which is a subset of process affinity mask. Initially it is the same as a process mask.

Affinity masks are pointers or bit vectors. To get and set these masks a set of APIs are used. Some of them are defined below:

```
BOOL GetProccessAffinityMask (
                                HANDLE hprocess,
                                LPDWORD lpProcessAffinityMask,
                                LPDWORD lpSystemAffinityMask);
```

It reads both the process and system affinity mask. On a single processor system, the value of masks will be 1.

```
BOOL SetProcessAffinityMask (
                                HANDLE hNamedPipe,
                                DWORD_PTR dwProcessAffinityMask);
```

The process affinity mask that is inherited by any child process can be set by this function. The new mask must be a subset of the mask returned by **GetProcessAffinityMask()**. The new value will affect all the threads in the process.

```
DWORD_PTR SetThreadAffinityMask (
                                HANDLE hThread,
                                DWORD dwThreadAffinityMask);
```

Further the thread masks are also set by a similar function. These functions are not designed consistently. **SetThreadAffinityMask()** returns a DWORD which is the previous mask while **SetProcessAffinityMask()** returns a **BOOL**.

Topic No - 181 - Interprocess Communication

Previously, we discussed how to create processes and how to create threads. Also, we discussed how to control concurrency. Now we will discuss how to pass information among processes. This is achieved via **Inter-process communication (IPC).** It is a mechanism that allows processes to communicate with each other. The communication between these processes can be seen as a method of co-operation between them.

Usually a file-like object called pipe can be used for IPC. There are two types.

1. **Anonymous Pipes**
2. **Named Pipes**

1. Anonymous Pipes

- Simple anonymous pipes are character based and half duplex.
- They are well suited for directing the output of one program into the input of another.

2. Named Pipes :

- Much more powerful than anonymous pipes
- They are full-duplex and message oriented.
- They allow network wide communication.
- There can be multiple open handles for a pipe.
- Transaction oriented named pipe functions make them suitable for client server applications.

Topic No - 182 - Anonymous Pipes

Anonymous pipes allow one-way (half-duplex) communication. They can be used to perform byte-based IPC. Each pipe has two handles: a read and a write handle.

CreatePipe() function can be used to create anonymous pipes:

```
BOOL CreatePipe (
                    PHANDLE phRead,
                    PHANDLE phWrite,
                    LPSECURITY_ATTRIBUTES lpsa,
                    DWORD cbPipe);
```

cbPipe is only a suggestion to pipe size. The default is set as 0.

Interprocess communication occurs between two processes. In this case one process can be the parent process and the other can be a child. Suppose the parent process wants to write output which the child will read. In this case parent will have to pass the *phRead handle to child. In other words, the **phRead** handle should belong to the child process and **phWrite** should belong to the parent process. This is usually done using the start-up structure as discussed previously.

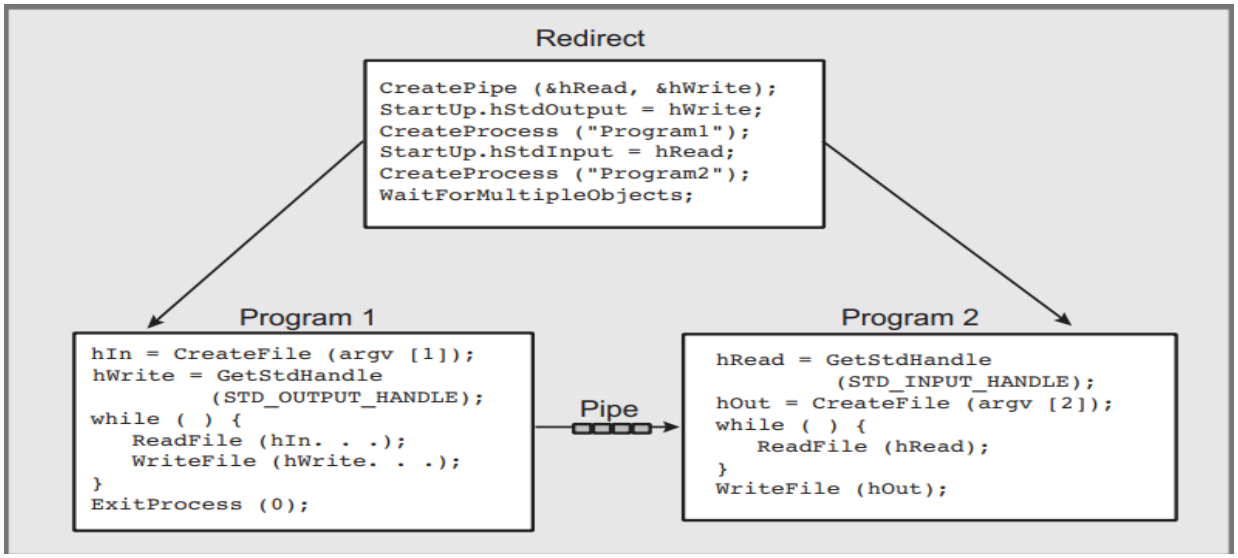
Reading a pipe read-handle will block, if the pipe is empty. Otherwise ReadFile() can read as many bytes as there are in the pipe or specified in ReadFile(). Similarly, a write operation will block if the pipe is in a buffer and the buffer is full.

Anonymous pipes work one way**. For two-way operation two pipes will be required**.

Topic 183: I/O Redirection using Anonymous Pipes

The given example is built in a way such that the parent process creates two child processes.

The Child processes are piped together. The parent process sets up the child processes in such a way such that their standard input and output can be redirected via the pipe. Child processes are designed such that they accumulate data and ultimately process it.



include "Everything.h"

int _tmain (int argc, LPTSTR argv [])

/* Pipe together two programs whose names are on the command line:

Redirect command1 = command2

where the two commands are arbitrary strings.

command1 uses standard input, and command2 uses standard output. Use = so as not to conflict with the DOS pipe. */

```
{
        DWORD i;
        HANDLE hReadPipe, hWritePipe; TCHAR command1 [MAX_PATH];
        SECURITY_ATTRIBUTES pipeSA = {sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
        /* Initialize for inheritable handles. */
        PROCESS_INFORMATION procInfo1, procInfo2;
        STARTUPINFO startInfoCh1, startInfoCh2;
        LPTSTR targv, cLine = GetCommandLine ();
        /* Startup info for the two child processes. */
         GetStartupInfo (&startInfoCh1);
        GetStartupInfo (&startInfoCh2);
        if (cLine == NULL)
                ReportError (_T ("\nCannot read command line."), 1, TRUE);
        targv = SkipArg(cLine, 1, argc, argv);
        i = 0;    /* Get the two commands. */
while (*targv != _T('=') && *targv != _T('\0')) {
                command1 [i] = *targv;
                targv++; i++;
        }
        command1 [i] = _T('\0'); if (*targv == _T('\0'))
                ReportError (_T("No command separator found."), 2, FALSE);
/* Skip past the = and white space to the start of the second command */
          targv++;
          while ( *targv != '\0' && (*targv == ' ' || *targv == '\t') ) targv++;
          if (*targv == _T('\0'))
                ReportError (_T("Second command not found."), 2, FALSE);
/* Create an anonymous pipe with default size.
The handles are inheritable. */
        if (!CreatePipe (&hReadPipe, &hWritePipe, &pipeSA, 0))
```

```
                ReportError (_T ("Anon pipe create failed."), 3, TRUE);
/* Set the output handle to the inheritable pipe handle, and create the first processes. */
        startInfoCh1.hStdInput = GetStdHandle (STD_INPUT_HANDLE);
        startInfoCh1.hStdError = GetStdHandle (STD_ERROR_HANDLE);
         startInfoCh1.hStdOutput = hWritePipe;
        startInfoCh1.dwFlags = STARTF_USESTDHANDLES;
if (!CreateProcess (NULL, command1, NULL, NULL,
TRUE,/* Inherit handles. */
0, NULL, NULL, &startInfoCh1, &procInfo1)) { ReportError (_T ("CreateProc1 failed."), 4, TRUE);
}
        CloseHandle (procInfo1.hThread);
        CloseHandle (hWritePipe);
/* Repeat (symmetrically) for the second process. */
        startInfoCh2.hStdInput = hReadPipe;
        startInfoCh2.hStdError = GetStdHandle (STD_ERROR_HANDLE);
        startInfoCh2.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
        startInfoCh2.dwFlags = STARTF_USESTDHANDLES;
if (!CreateProcess (NULL, targv, NULL, NULL,
TRUE,/* Inherit handles. */
0, NULL, NULL, &startInfoCh2, &procInfo2))
                ReportError (_T ("CreateProc2 failed."), 5, TRUE);
        CloseHandle (procInfo2.hThread);
        CloseHandle (hReadPipe);
/* Wait for both processes to complete.
The first one should finish first, although it really does not matter. */
        WaitForSingleObject (procInfo1.hProcess, INFINITE);
        WaitForSingleObject (procInfo2.hProcess, INFINITE);
        CloseHandle (procInfo1.hProcess);
        CloseHandle (procInfo2.hProcess);
        return 0;
}
```

Topic 184: (Named Pipes)

Named pipes are the general-purpose mechanism for implementing IPC-based applications like networked file access and client/server systems.

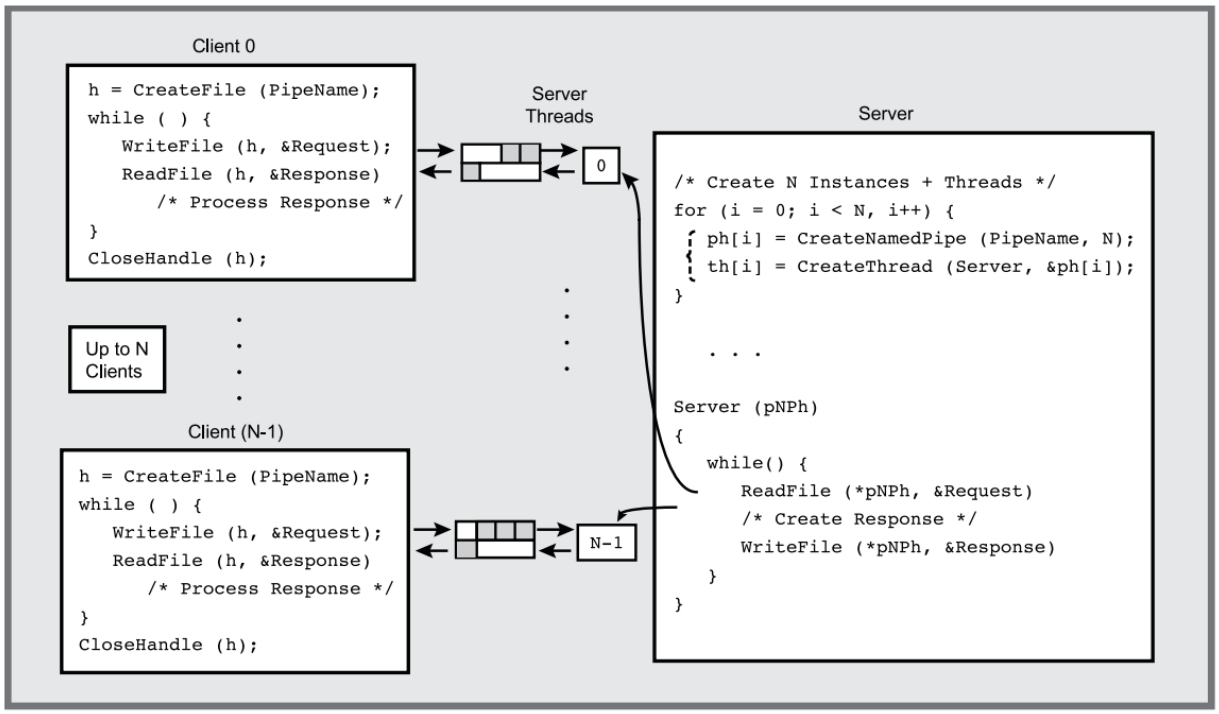
Anonymous pipes are simplistic; however, they have many disadvantages.

Features of Named Pipes:

- Named pipes are a powerful tool for IPC based applications.
- Named pipes are message-oriented hence the reading process can read varying length messages precisely as sent by the writing process.
- Named pipes are bi-directional. It means that two processes can exchange messages over the same named pipe.
- There can be multiple independent instances of pipes with the same name. For example, several clients can communicate with a server concurrently using distinct instances of named pipes. Each client can have its own distinct named pipe and the server can respond to the client using the same pipe.
- Networked client can access the pipe by name. Named pipe communication is the same whether the two processes are on the same machine or on different machines.
- Several convenience and connection functions greatly simplify named pipes operation which otherwise might be complex such as request/response and client/server operation.

Topic 185: (Using Named Pipes)

CreateNamedPipe() creates an instance of the named pipe and returns a handle. The function also specifies the maximum number of instances and hence the number of clients the pipe can support. Normally the creating process is regarded as a server. Client processes open pipe with **CreateFile**.



The above figure shows an illustrative client/server relationship, and the pseudocode shows the scheme for using named pipes.

In this figure, the server creates multiple instances of the same pipe, all of them can support a client. The server also creates a thread for each named pipe instance, so that each client has a dedicated thread and named pipe instance.

Topic 186: (Creating Named Pipes)

CreateNamedPipe() creates an instance of the named pipe and returns a handle. here is the specification of this function:

```
HANDLE CreateNamedPipe (
                    LPCTSTR lpName,
                    DWORD dwOpenMode,
                    DWORD dwPipeMode,
                    DWORD nMaxInstances,
                    DWORD nOutBufferSize,
                    DWORD nInBufferSize,
                    DWORD nDefaultTimeOut,
                    LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Details of Parameters:

- **lpName** indicates the pipe name which must be of the form.

  \\.\pipe\pipeName

  The period (.) stands for local machine. Creating a pipe on a remote machine is not possible. Name is case insensitive and can be upto 256 characters. It can contain any character except backslash.

- **dwOpenMode** is the open mode which can be.
  - ✓ **PIPE_ACCESS_DUPLEX,**
  - ✓ **PIPE_ACCESS_INBOUND,**
  - ✓ **PIPE_ACCESS_OUTBOUND** along with many other options.
- **dwPipeMode** indicates whether writing is message oriented or byte oriented.
- **nMaxInstance** determines maximum number of pipes instances.
- **nOutBufferSize** and nInBufferSize give the size in bytes of input and output buffer.
- **nDefaultTimeOut** is the default timeout period.
- **lpSecurityAttributes** are the security attributes as discussed previously.

First call to the **CreateNamedPipe()** function will create a named pipe and an instance. Closing the last handle to an instance will delete the instance and the named pipe.

Topic 187: (Named Pipes Client Connection) (2 minutes video)

A client connects to a named pipe using **CreateFile** with the pipe name. In many cases, the client and server are on the same machine.

The pipe name would be like this:

\\.\pipe\[path]pipename

If the server is on a different machine, the name would take this form:

\\servername\pipe\pipename

Using the server's name as (.) when the server is local rather than using the local machine name delivers significantly better connection performance.

Topic 188: (Named Pipe Status Function) (2 minutes video)

There are seven functions to interrogate pipe status information. Some functions are also used to set the state information. They are mentioned briefly:

- **GetNamedPipeHandleState()**

Returns information whether the pipe is in blocking or non-blocking mode, message oriented or byte oriented, number of pipe instances, and so on.

- **SetNamedPipeHandleState()**

Allows the program to set the same state attributes. Mode and other values are passed as reference so that NULL can also be passed indicating no change is desired.

- **GetNamedPipeInfo()**

Determines whether the handle is for client or a server, buffer sizes, and so on.

- Some functions get information regarding client name, client and server session ID and process ID such as **GetNamedPipeClientSessionId(), GetNamedPipeServerProcessId()**

Topic 189: (Named Pipe Connection Functions) (2 minutes video)

The server creates a named pipe instance. Once a pipe is created the server would wait for a client to connect. The connection is established using the **ConnectNamedPipe()** function

This function is explained below:

```
BOOL ConnectNamedPipe (
                        HANDLE hNamedPipe,
                        LPOVERLAPPED lpOverlapped)
```

- If **lpOverLapped** is set to NULL, the **ConnectNamedPipe()** will return as soon as there is a client connection. On Connection it returns a TRUE.
- It returns a FALSE if the pipe is already connected with error **ERROR_PIPE_CONNECTED**.
- Once the client is connected the server can read and write messages using **ReadFile()** and **WriteFile()**.
- Call **DisconnectNamedPipe()** to disconnect from the handle.
- **WaitNamedPipe()** is used to synchronize connections to the server.

Topic 190: Client Server Named Pipe Connection

The connection sequences for the Server are as follows:

- ✓ Server makes a client connection.
- ✓ Communicates with the client.
- ✓ Ultimately the client disconnects.
- ✓ As a result ReadFile() returns FALSE.
- ✓ The server-side connection is disconnected.
- ✓ Server may now connect with another client.

```
/* Named pipe server connection sequence. */
hNp = CreateNamedPipe ("\\\\.\\pipe\\my_pipe", ...);
while (... /* Continue until server shuts down. */) {
      ConnectNamedPipe (hNp, NULL);
      while (ReadFile (hNp, Request, ...) {
         ...
         WriteFile (hNp, Response, ...);
      }
      DisconnectNamedPipe (hNp);
}
CloseHandle (hNp);
```

The connection sequences for the Client are as follows:

- ✓ The client connects with a server.
- ✓ Communicates with the server using **CreateFile()**
- ✓ Performs Read and Write operations and ultimately disconnects.
- ✓ Now the connection is available for another client to connect.

```
/* Named pipe client connection sequence. */
WaitNamedPipe ("\\\\ServerName\\pipe\\my_pipe",
         NMPWAIT_WAIT_FOREVER);
hNp =
         CreateFile ("\\\\ServerName\\pipe\\my_pipe", ...);
while (... /* Run until there are no more requests. */ {
      WriteFile (hNp, Request, ...);
      ...
      ReadFile (hNp, Response);
}
CloseHandle (hNp); /* Disconnect from the server. */
```

==========

THE END

==========

Changes Made in Edition 3:

- ✓ Improved Formatting
- ✓ Used Different colors in the handouts.
- ✓ APIs are written in box having light grey color making it easier to understand.
- ✓ Written API's parameters explanation in a better formatting.
- ✓ Highlighted the handouts by myself.
- ✓ Highlighted MCQ's and mentioned the mcq's statements and answer too.
- ✓ Highlighted the short and long questions and also mentioned their Question statements too.

Note: If you find any error in the handouts, Kindly contact me on 0301-7923536 WhatsApp.