CS609

SYSTEM PROGRAMMING

93 To 130

Arranged By Amna

Topic 93: Process Identities

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

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

HANDLE GetCurrentProcess (VOID)

DWORD GetCurrentProcessId(VOID)

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

HANDLE OpenProcess(DWORD dwDesiredAccess,

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);

hSourceProcessHandle is the source process handle. 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. 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.

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

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

i. DWORD waitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
ii. DWORD waitForMultipleObjects(DWORD nCount, const HANDLE*lpHandles, BOOL bWaitAll,DWORD dwMilliseconds);

- hHandle is the single object for which the process waits.
- lpHandle is the array of handles for which the process waits.
- nCount is the number of objects in an array. Should not exceed MAXIMUM WAIT OBJECTS
- dwMilliseconds is the timeout period for wait. 0 for no wait and INFINITE for indefinite wait.
- 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 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);

And

BOOL SetEnviromentVariable(LPCTSTR ipName, 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

it can be useful to limit some processes right like giving PROCESS_TERMINATE 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

it can be useful to limit some processes right like giving PROCESS_TERMINATE right to parent process only.

Topic No.98

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;
                startUpSearch.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

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

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 hread handle.

Topic No.101

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

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_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.103

Generating Console Events

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);

dwCtrlEvent can be CTRL_C_EVENT or CTRL_BREAK_EVENT

dwProcessGroupId is the id of the process group

Topic No.104

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

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;
}
```

Topic 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 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);

hJob is handled to the existing job object

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.

lpJobObjectInformation is the structure congaing the actual information as it uses a different structure for each class

Topic 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.

```
/* 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 109

Threads 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 by the use of threads.

The conventional scenario uses single-threaded processes being run concurrently

This scenario has varying disadvantages

Switching among processes is time-consuming and expensive. Threads can easily allow concurrent processing of the same function hence reducing overheads

Usually, independent processes are not tightly coupled, and hence sharing resources is difficult.

Synchronization and mutual exclusion in single-threaded processes halt computations.

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 110

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 111

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.

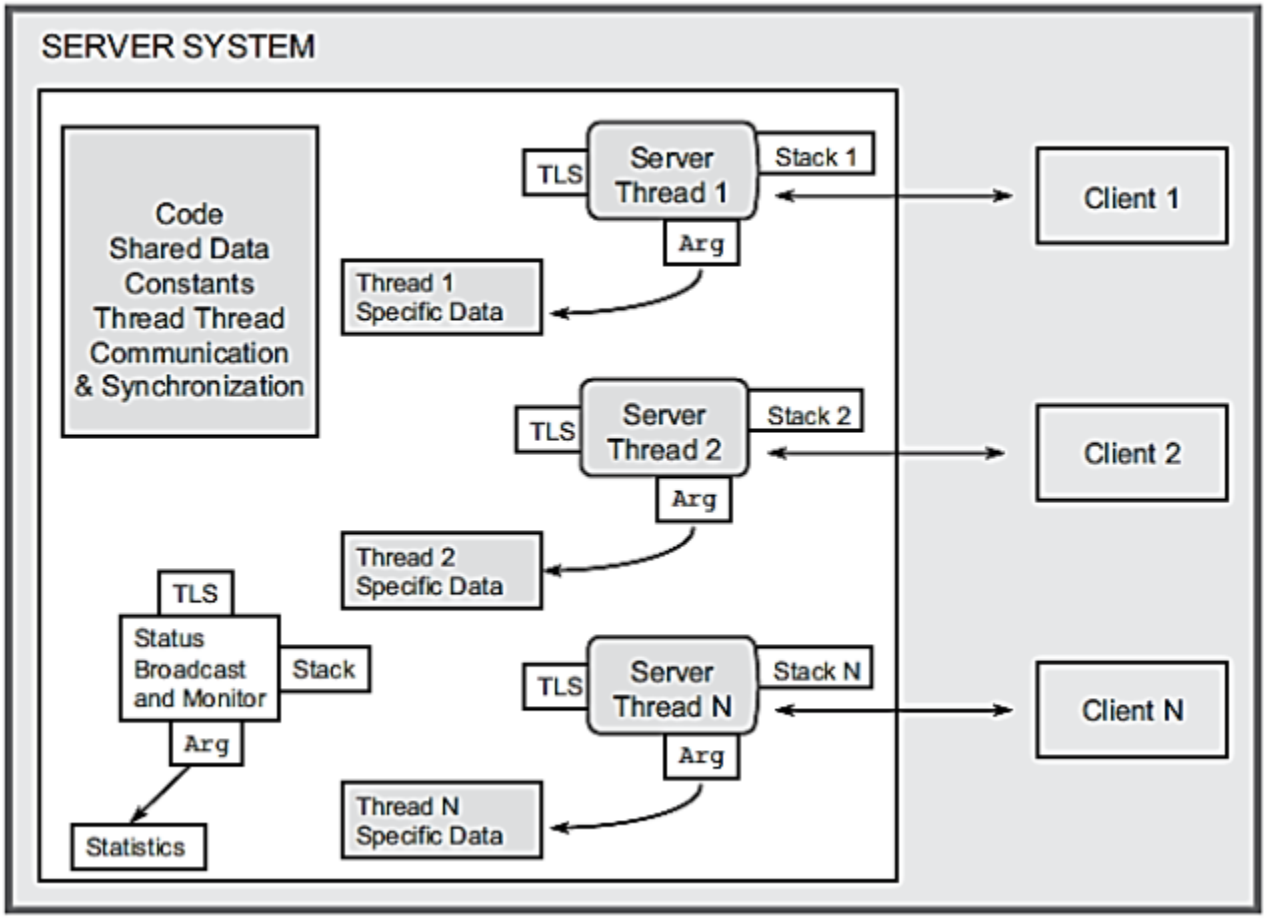
Moreover, each thread has its own stack for function calls.

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 112

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);

lpsa is security attributes structure used previously.

dwStackSize is the stack size in bytes.

lpStartAddress is the starting address of the thread function within the calling process of the form:

DWORD WINAPI ThreadFunc(LPVOID)

The function returns a DWORD value which is usually an exit code and accepts a single pointer argument.

lpThreadParam is the pointer passed to the thread and is usually interpreted as a pointer to a structure containing arguments

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.

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 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 reffering to the thread are invalidated.

If the thread is linked to some DLL then the DllMain() function is invoked with te 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 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 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 to pause 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)

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

Topic 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 nonsignaled.

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 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.

Topic 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 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

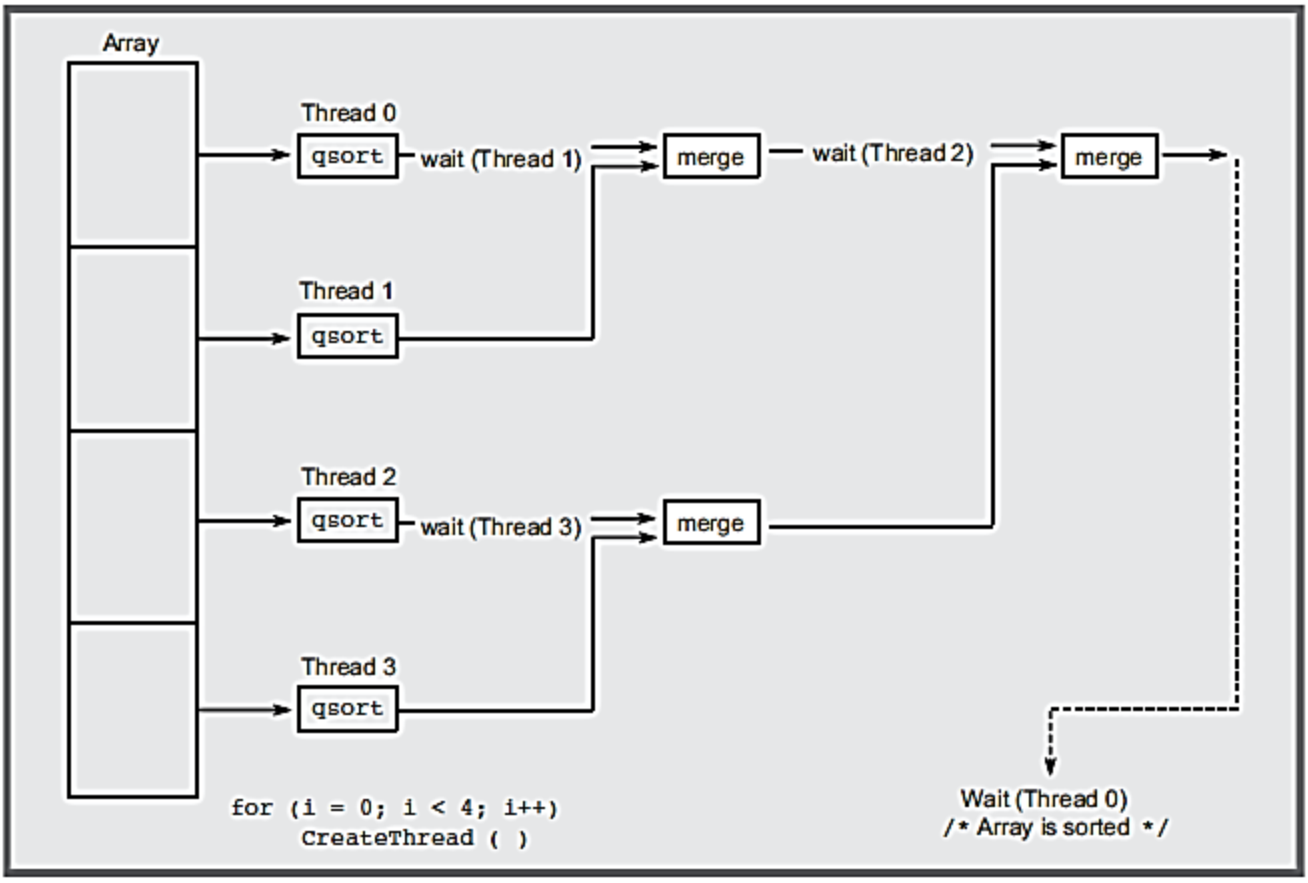
There are many advantages to using these models when designing a multithreaded program, including the following.

- 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 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 121

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 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

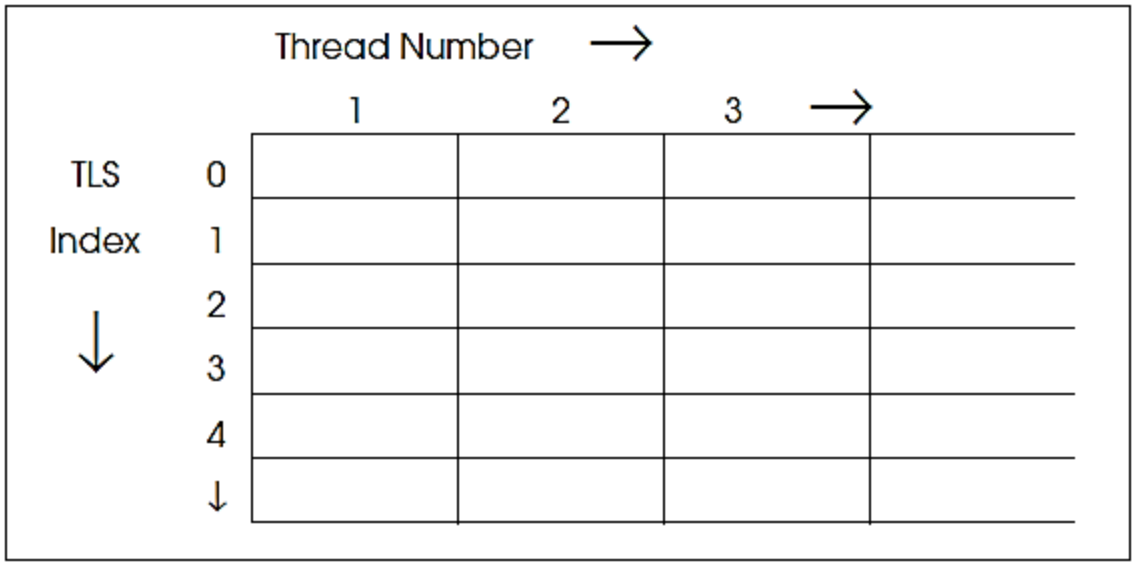
multiprocessors as well. If we have a system with multiprocessors, we should learn to program to use the potential of multiprocessors because the processor's speed has reached its bottleneck i.e. after a certain limit, its speed cannot be enhanced. Parallelization is the key to future performance improvement 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 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)**: Frees the specified index.

- **LPVOID TlsGetValue (DWORD dwTlsIndex) and BOOL TlsSetValue (DWORD dwTlsIndex, LPVOID lpTlsValue):** 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

- 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 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 **BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority) and DWORD GetPriorityClass(HANDLE hProcess)** respectively.

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)** AND **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 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.

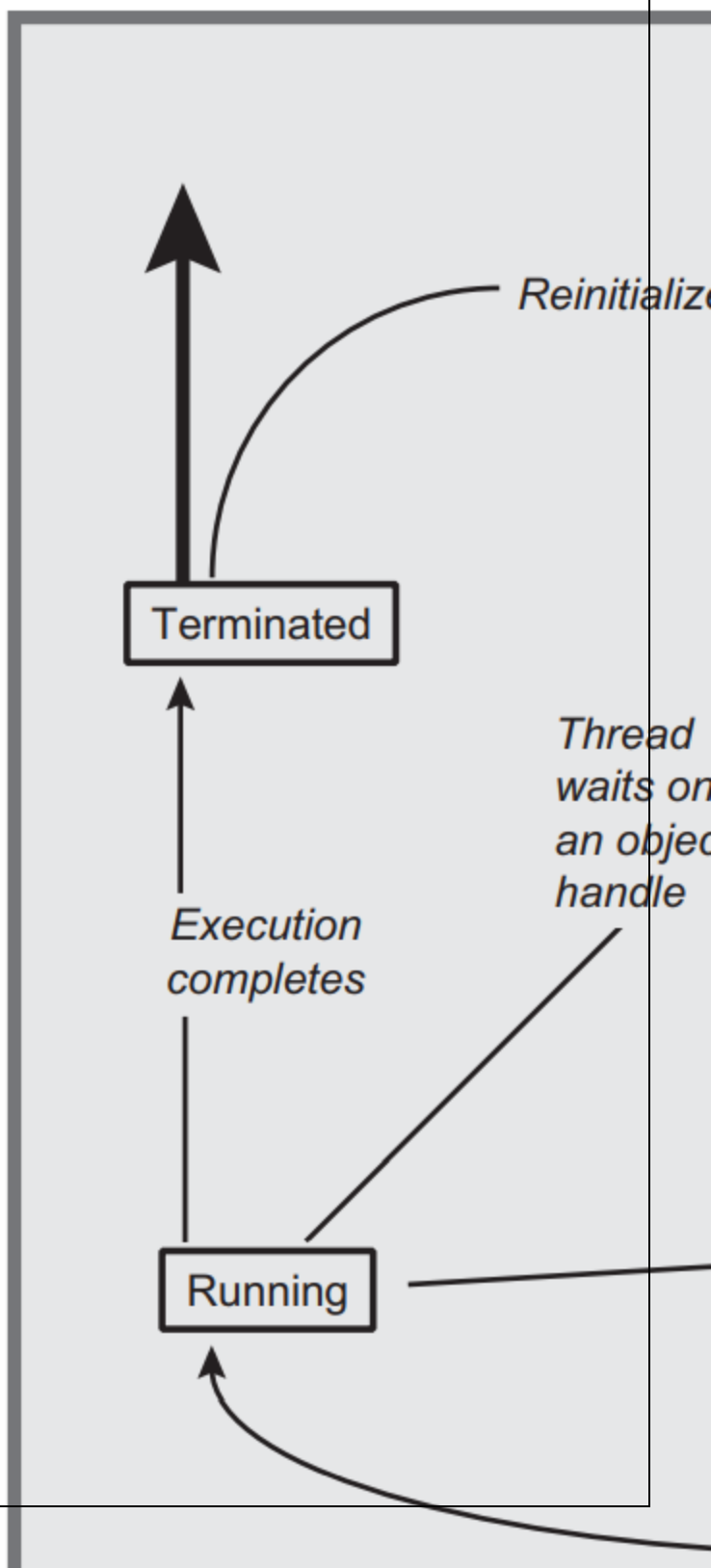

Reinitializ
Terminated
Thread
waits on
an objec
handle
Execution
completes
Running

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 126

- 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 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 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 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.

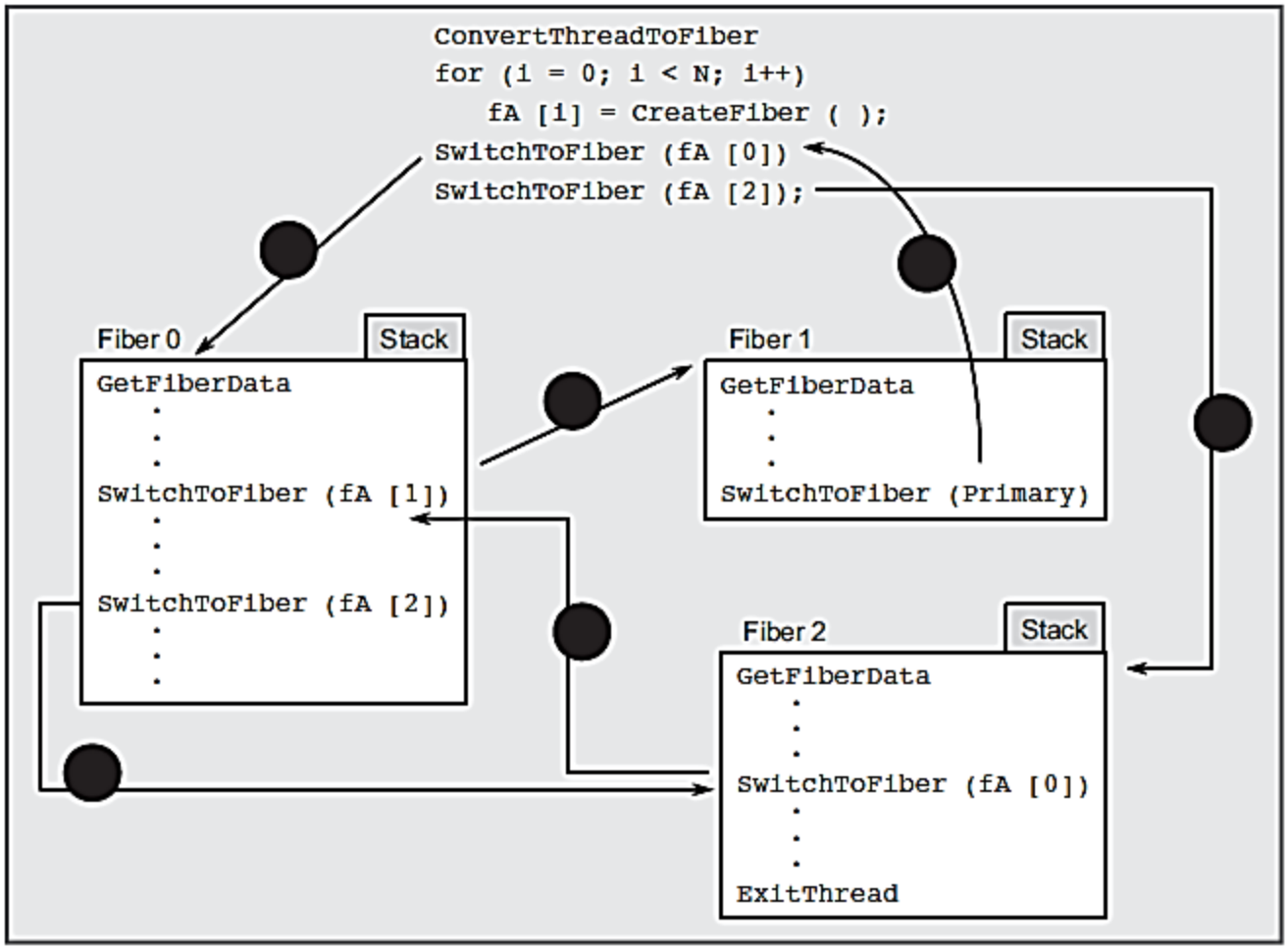
An existing fiber can be deleted using

DeleteFiber()

Topic 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)