Thread Manager: Introduction, Theory and Implementation

Romil Garg
Mindfire Solutions
www.mindfiresolutions.com

Introduction

It is required to design and implement a thread management application, which is supposed to be integrated with the existing architecture of a telecommunications company for monitoring 96 telephone ports and responding to the messages received by firing corresponding application. The application's purpose is to manage the communication between these 96 threads, which are responsible for monitoring the telephone ports.
To work effectively with Windows NT system services, we must have a solid understanding of Kernel Objects, Processes and Threads. Under Windows NT, an instance of an executing application is a Process and Threads are at the heart of the Windows NT multitasking abilities. We'll see how these processes and threads are created, destroyed and the various attributes associated with them.

Kernel Objects, Processes and Threads

Kernel Objects

They are used by the system and the applications we write to manage numerous resources such as Processes, Threads and Files. The system creates and manipulates several types of kernel objects:

a. Event objects
b. File – mapping objects
c. File objects
d. Mailslot objects
e. Mutex objects
f. Pipe objects
g. Process objects
h. Semaphore objects
i. Thread objects

- Each kernel object is just a memory block allocated by the kernel and is accessible only by the kernel.
- This memory block is a data structure whose members maintain information about the object.
- Kernel objects are owned by the kernel, not by a process i.e. if your process calls a function that creates a kernel object and then your process terminates, the kernel object is not necessarily destroyed.
- All functions that create kernel objects return process-relative handles that can be used successfully by any and all threads that are running in the same process.

Process

A process is usually defined as an instance of a running program.

- In Win32, a process owns a 4 GB address space.
- Win32 process executes nothing – it simply owns a 4 GB address space that contains the code and data for an application's EXE file. Any DLLs required by the EXE also have their code and data loaded into the process's address space.
- A process owns certain resources like – Files, Threads, and Dynamic Memory Allocations.
- The various resources created during a process's life are destroyed when the process is terminated.
- For a process to accomplish anything, the process must own a thread.
- It is this thread, which is responsible for executing the code contained in the process's address space.
- A Single Process might contain several threads all of them executing code "simultaneously" in the process's address space.
- To do this, each thread has its very own set of CPU registers and its own stack.

- For all the threads to run, the O.S. schedules some CPU time for each individual thread.

Steps in Creating a Process

1. Application calls *CreateProcess* function.
2. System creates a process kernel object with an initial usage count of 1.
3. It is a small data structure that the O.S. uses to manage the process.
4. System then creates a virtual 4 GB address space for the new process.
5. It then loads the code and data for the EXE file and any required DLLs into the process's 4 GB address space.
6. System then creates a Thread Kernel Object (with a usage count of 1) for the new process's primary thread.
7. Thread Kernel Object is a small data structure that the O.S. uses to manage the thread.
8. The primary thread will begin by executing the C run-time start-up code, which will eventually call your WinMain function.
9. If the system successfully creates the new process and primary thread, *CreateProcess* returns TRUE.

Process Termination

A Process can be terminated in 3 ways:

1. One thread in the process calls the *ExitProcess* function.
2. A thread in another process calls the *TerminateProcess* function.
3. All the threads in the process just die on their own.

Thread

A thread describes a path of execution within a process.

- Every time a process is initialized, the system creates a primary thread.
- The whole idea behind creating additional threads is to utilize as much of the CPU's time as possible.
- Each thread is allocated its very own stack from the owning process's 4 GB address space.
- Local and Automatic variables are created on the thread's stack and are less likely to be corrupted by another thread.
- AVOID the use of Static and Global variables, as multiple threads can access the variables at the same time, potentially corrupting the variable's contents.

Thread's Context Structure

- Each thread has its own set of CPU registers, called the thread's context.
- This context structure reflects the state of the thread's CPU registers when the thread was last executing.
- When a thread is scheduled CPU time, the system initializes the CPU's registers with the thread's context.
- The CPU registers also include a stack pointer that identifies the address of the thread's stack.

CreateThread Function

This function is used to create additional threads from the Primary Thread of the process. For every call to CreateThread, the system must perform the following steps:

- Allocate a thread kernel object to identify and manage the newly created thread. This object holds much of the system information to manage the thread. A handle to this object is the value returned from the CreateThread function.
- Initialize the thread's exit code (maintained in the thread kernel object) to STILL_ACTIVE and set the thread's suspend count to 1.
- Allocate a Context structure for the new thread.
- Prepare the thread's stack by reserving a region of address space, committing 2 pages of physical storage to the region, setting the protection of the committed storage to PAGE_READWRITE, and setting the PAGE_GUARD attribute on the second-to-top page.
- The lpStartAddr and lpvThread values are placed on the top of the stack so that they look like parameters passed to the StartOfThread function.
- Initialize the stack pointer register in the thread's context structure to point to the values placed on the stack in Step5; initialize the instruction pointer register to point to the internal StartOfThread function.

Suspending and Resuming Threads

- A thread can be created in a suspended state by passing the CREATE_SUSPENDED flag to CreateProcess or CreateThread function.
- A thread can also be suspended by calling the function:
- DWORD SuspendThread(HANDLE hThread)
- By doing this, the thread object is given an initial suspend count of 1, which means the system will never assign a CPU to execute the thread.
- To allow the thread to begin execution, another thread must call *ResumeThread* and pass it the thread handle returned by the call to *CreateThread*.
- A single thread can be suspended several times.
- If a thread is suspended 3 times, the thread must be resumed 3 times before it is eligible for assignment to a CPU.

Terminating a Thread

A thread can be terminated in 3 ways:

1. A thread kills itself by calling the *ExitThread* function.
    - This function terminates the thread and sets the thread's exit code to fuExitCode.
    - It does not return a value, because the thread has terminated.

2. A thread in the same or in another process calls the *TerminateThread* function.
    - The function ends the thread identified by the hThread parameter and sets its exit code to dwExitCode.
    - This function exists so that a thread can be terminated when it no longer responds.

3. The process containing the thread terminates.
    - The *ExitProcess* and *TerminateProcess* functions terminate all the threads contained in the process being terminated.

Important Note

- When a thread dies by calling *ExitThread*, the stack for the thread is destroyed. However, if the thread is terminated by *TerminateThread*, the system does not destroy the stack until the process that owns the thread terminates. This is because other threads might still be using pointers that reference data contained on the terminated thread's stack.

- When a thread ends, the system notifies any DLLs attached to the process owning the thread that the thread is ending. BUT, if *TerminateThread* is called, the system doesn't notify any DLLs attached to the process, which can mean that the process won't be closed down correctly.

Miscellaneous

- A thread can get the handle of the process it is running in by calling *GetCurrentProcess.*
  HANDLE GetCurrentProcess(VOID)
  This function returns a pseudo-handle to the process; it doesn't create a new handle, and it doesn't increment the process object's usage count.

- A thread can acquire the ID of the process it is running in by calling *GetCurrentProcessId*.
  DWORD GetCurrentProcessId(VOID)
  This function returns the unique, systemwide ID that identifies the process.

- For a thread to acquire a handle to itself, it must call:
  HANDLE GetCurrrentThread(VOID)

- A thread acquires its ID by calling:
  DWORD GetCurrentThreadId(VOID)

Whenever multiple threads are executing simultaneously or are being preemptively interrupted, an application will often need to suspend a thread to prevent data corruption. Windows NT offer several objects for performing thread synchronization, all of which are discussed in the following section.

Thread Synchronization

- Synchronizing resource access between threads is a common problem when writing multithreaded applications. Having two or more threads simultaneously accessing the same data can lead to undesirable and unpredictable results.
  For example, one thread could be updating the contents of a structure while another thread is reading the contents of the same structure. It is unknown what data the reading thread will receive: the old data, the newly written data, or possibly a mixture of both.

- Synchronization is used when access to a resource must be controlled to ensure integrity of the resource. Synchronization objects are used to gain access to these controlled resources.

- Five main synchronization objects are:

  1. Critical Sections
  2. Mutexes
  3. Semaphores
  4. Events
  5. Waitable Timers

- A thread synchronizes itself with another thread by putting itself to sleep.

- When the thread is sleeping, the O.S. no longer schedules CPU time for it, and it therefore stops executing.

- Before the thread puts itself to sleep, it tells the O.S. what "special event" has to occur for the thread to resume execution.

Critical Sections

- A critical section is a small section of code that requires exclusive access to some shared data before the code can execute.
- Simplest to use.
- It can be used to synchronize threads only within a single process.
- They allow only one thread at a time to gain access to a region of data.
- Critical sections are simply global variables.

Semaphores

- Semaphore kernel objects are used for resource counting.
- They offer a thread the ability to query the no. of resources available – If one or more resources are available, the count of available resources is decremented.
- Because several threads can affect a semaphore resource count, a semaphore, unlike a critical section or a mutex, is NOT considered to be owned by a thread.

Waitable Timers

- Threads can be put to sleep until an object becomes signaled. If a thread in a parent process needs to wait for the child process to terminate, the parent's thread puts itself to sleep until the kernel object identifying the child process becomes signaled.
- Threads use two main functions to put themselves to sleep while waiting for kernel objects to become signaled:
    1. **DWORD WaitForSingleObject(HANDLE** *hObject***, DWORD** *dwTimeout***)**
    2. **DWORD WaitForMultipleObjects(DWORD** *cObjects***, LPHANDLE** *lpHandles***, BOOL** *bWaitAll***, DWORD** *dwTimeout***)**

- The *WaitForSingleObject* function tells the system that the thread is waiting for the kernel object identified by the hObject parameter to be signaled. If the specified kernel object does not become signaled in the specified time, the system should wake up the thread and allow it to continue executing.
- The *WaitForMultipleObjects* function waits either for several objects to be signaled or for one object from a list of objects to be signaled.

Mutexes

- They can be used to synchronize data access across multiple processes.
- To do this, a thread in each process must have its own process-relative handle to a single mutex object.
- Mutex objects are owned by a thread.
- Mutex objects in addition to being signaled or non-signaled, remember which thread owns them.

- A Mutex is abandoned if a thread waits for a mutex object, grabs the object (putting it in the non-signaled state), and then terminates.
- To use a mutex, one process must first create the mutex with the CreateMutex function.

  HANDLE CreateMutex(LPSECURITY_ATTRIBUTES *lpMutexAttributes***, BOOL** *bInitialOwner***, LPCTSTR** *lpName***)**

- bInitialOwner parameter indicates whether the thread creating the mutex should be the initial owner of the mutex. The value TRUE means that the thread will own the mutex and any thread that waits on the mutex will be suspended until the thread that created the mutex releases it. The value FALSE means that the mutex is not owned by any thread and is therefore created in the signaled state.
- The first thread to wait for the mutex will immediately gain ownership of the mutex and continue operation.
- *ReleaseMutex* is the function that changes the mutex from the non-signaled state to the signaled state. This function has an effect only if the thread that is calling *ReleaseMutex* also has the ownership of the mutex. Immediately after this function is called, any thread that is waiting for the mutex can grab hold of it and begin executing.

Events

- Events are used to signal that some operation has completed
- Events are most commonly used when one thread performs initialization work and then, when it completes, signals another thread to perform the remaining work.
- Events are created using the function:

  HANDLE CreateEvent(LPSECURITY_ATTRIBUTES *lpEventAttributes***, BOOL** *bManualReset*, **BOOL** *bInitialState***, LPCTSTR** *lpName***)**

- Events are of two types:
  1. Manual-Reset Events – It is used to signal several threads simultaneously that an operation has completed.
  2. Auto-Reset Events – It is used to signal a single thread that an operation has completed.

- In Manual-Reset Events, we might have several threads, all of them waiting for the same event to occur. When this event is signaled, all threads waiting on the event are allowed to run.
- Threads in other processes can gain access to the object by calling *CreateEvent* using the same value in the lpName parameter or by calling the function *OpenEvent.*
- Events are closed by calling the *CloseHandle* function.

After going through Processes, Threads and Thread Synchronization, our next step is to see what are Dynamic Link Libraries (DLLs). DLLs have always been the cornerstone of all Windows applications. In the following section we'll see how a DLL is managed and how it is mapped into a process's address space.

Dynamic Link Library

Basic Concepts

A Windows program is an executable file that generally creates one or more windows and uses a message loop to receive user input. Dynamic link libraries are generally not directly executable, and

they do not receive messages. They are separate files containing functions that can be called by programs and other DLLs to perform certain computations or functions.

- A DLL is simply a set of source code modules, with each module containing a set of functions that an application (executable file) or another DLL will call.
- In the Win32 environment, a DLL must be mapped into the process's address space before an application can successfully call functions in the Dll.
- In order for a thread to call a function in a DLL, the DLL's file image must be mapped into the address space of the calling thread's processes.

Mapping a DLL into a Process's Address Space

It can be accomplished in two ways: Implicit Linking to functions in the DLL and Explicitly loading the DLL.

Implicit Linking

- Implicit linking is the most common method for mapping a DLL's file image into a process's address space.
- While linking an application, you must specify a set of LIB files to the linker. Each LIB file contains the list of functions that a DLL file is allowing an application (or another DLL) to call.
- When the linker sees that the application is calling a function that is referenced in a DLL's LIB file, the linker embeds information into the resultant EXE file image that indicates the name of the DLL containing the function that the EXE requires.
- When searching for the DLL, the system looks for the file image in the following locations:
    1. The directory containing the EXE file image.
    2. The process's current directory.
    3. The Windows system directory.
    4. The Windows directory.
    5. The directories listed in the PATH environment variable.

- When using this method, any DLL file images mapped into the process's address space are not unmapped until the process is terminated.

Explicit Linking

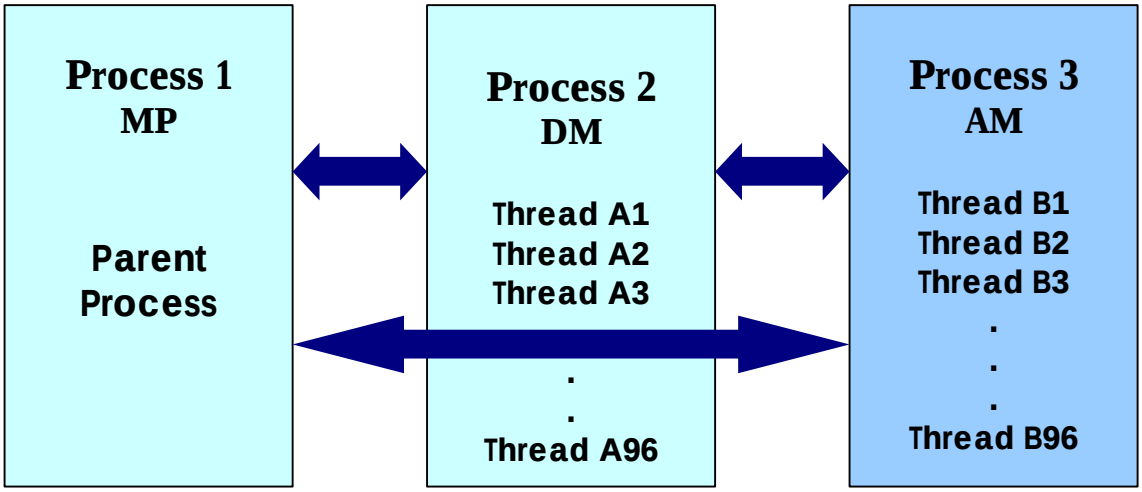
- A DLL's file image can be explicitly mapped into a process's address space when one of the process's threads calls either the *LoadLibrary* or the *LoadLibraryEx* function:
- Both of these functions locate a file image on the user's system and attempt to map the DLL's file image into the calling process's address space.
- Both these functions return a HINSTANCE value which identifies the virtual memory address where the file image was mapped.
- If the DLL could not be mapped into the process's address space, the function returns NULL.

Thread Manager : Proposed Application

Problem Description

It is required to design and implement a thread management application. The application's purpose is to manage the communication between threads, which have been created according to a specific architecture. A detailed description is presented below.

Design of the proposed application



The parent process creates process 2, which in turn spawns threads A1 to A96 in process 2's address space. Thereafter, each of these threads may spawn a corresponding thread B1 to B96 in another process's address space (process 3). Process 3 may possibly be running on a remote CPU.

Thread Ax monitors the health of thread Bx. This is done through a *Heartbeat* mechanism. After a fixed duration which may be specified Bx sends an 'OK' signal to Ax . This is a single heartbeat. If Ax does not receive a heartbeat from Bx for over a specified limit, it understands that Bx is in trouble. In such a situation, the corrective action it takes is to kill Bx.

In addition to health monitoring, Ax and Bx need to communicate with each other. This communication is asynchronous i.e. Ax may post a message to Bx at any time and as such, is implemented through a message queue. Ax and Bx have methods to post a message to each other's queues and to read from their respective message queues.

Thread Ax also send heartbeats to the parent process. The parent process monitors heartbeats from Ax and kills and re-spawns an Ax that it considers to be unwell. The parent process and Ax communicate among themselves too. The parent process can post messages to Ax's queue and read messages posted by Ax to its own queue.

Further, two way communication between the parent process and Bx is also there which is similar to the Ax to Bx and parent process to Ax communication defined above.

Test method

In order to be able to stress test the framework, the application allows the following to be specified by the user:

1. Heartbeat intervals for Bx - Ax and Ax - parent process channels.
2. Maximum permissible duration that can elapse without heartbeat for the two channels.
3. Messaging intensity for Bx - Ax, Ax - parent process, Bx - parent process channels.
4. Messaging intensity for Ax - Bx, parent process - Ax, parent process - Bx channels.
5. Number of Ax threads created.
6. Probability that a Message arrives on Ax port.

Creating Threads

There are 3 functions for creating threads which were taken into consideration. They are:

1. CreateThread
2. _beginthread
3. _beginthreadex

I CreateThread

HANDLE CreateThread(
LPSECURITY_ATTRIBUTES *lpThreadAttributes***,** //pointer to security attributes
DWORD *dwStackSize***,** // initial thread stack size
LPTHREAD_START_ROUTINE *lpStartAddress***,** //pointer to thread func
LPVOID *lpParameter***,** // argument for new thread
DWORD *dwCreationFlags***,** // creation flags
LPDWORD *lpThreadId* // pointer to receive thread ID
)

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process. If the function succeeds, the return value is a handle to the new thread. If the function fails, the return value is NULL. The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the DWORD return value is used to terminate the thread in an implicit call to the **ExitThread** function. The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle.**

A thread that uses functions from the C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

II _beginthread

unsigned long _beginthread(
void(__cdecl **start_address* **)(void *),** //Start address of routine that begins execution of new thread
unsigned *stack_size***,** //Stack size for new thread or 0
void **arglist* //Argument list to be passed to new thread or NULL
)

To use **_beginthread**, the application must link with one of the multithreaded C run-time libraries. If successful, it returns a handle to the newly created thread.

The **_beginthread** function creates a thread that begins execution of a routine at *start_address*. The routine at *start_address* must use the **__cdecl** calling convention and should have no return value. When the thread returns from that routine, it is terminated automatically.

You can call **_endthread** explicitly to terminate a thread; however, **_endthread** is called automatically when the thread returns from the routine passed as a parameter. Terminating a thread with a call to **endthread** helps to ensure proper recovery of resources allocated for the thread.

III _beginthreadex

unsigned long _beginthreadex(
void **security***,** //Security descriptor for new thread
unsigned *stack_size***,** //Stack size for new thread or 0
unsigned (__stdcall **start_address* **)(void *),** //Start address of routine that begins execution of new thread
void **arglist***,** //Argument list to be passed to new thread or NULL
unsigned *initflag***,** //Initial state of new thread
unsigned **thrdaddr* //Points to a 32-bit variable that receives the thread identifier
)

_beginthreadex resembles the Win32 **CreateThread** API more closely than does **_beginthread**. **_beginthreadex** differs from **_beginthread** in the following ways:

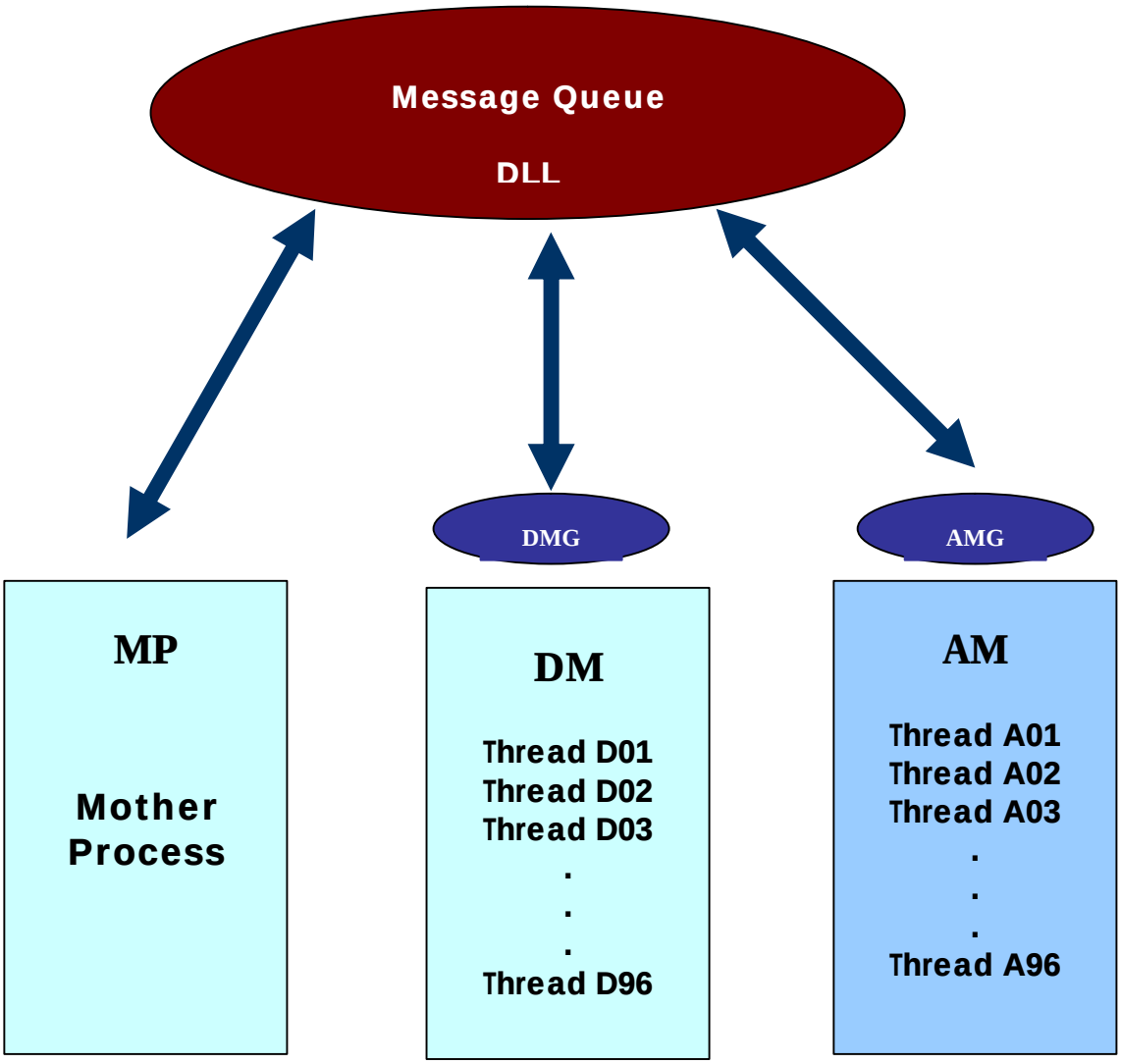
- **_beginthreadex** has three additional parameters: *initflag*, *security*, *threadaddr*. The new thread can be created in a suspended state, with a specified security (Windows NT only), and can be accessed using *thrdaddr*, which is the thread identifier.
- The routine at *start_address* passed to **_beginthreadex** must use the **__stdcall** calling convention and must return a thread exit code.
- **_beginthreadex** returns 0 on failure, rather than –1.
- A thread created with **_beginthreadex** is terminated by a call to **_endthreadex**.

- **_beginthread** is used to create the threads because of the following reasons:
    - Since we are using C run-time libraries, using **CreateThread** & **ExitThread** resulted in memory leaks and abnormal program termination.
    - The three additional parameters offered by **_beginthreadex** are not required as threads have to be created in running state and security descriptor & thread identifier are of no use.
    - **_endthread** automatically closes the thread handle whereas **_endthreadex** does not.
- **_endthread** is used to terminate threads forcefully which automatically closes the thread handle and helps to ensure proper recovery of resources allocated for the thread.

Problem: While creating 96 threads in DM, thread name has been passed as parameter to the thread routine, which executes the new thread. After going to this routine, this parameter doesn't retain its value as in the mean time a new thread has been created and thus the value of this argument changes.

Solution: For this problem, before a new thread is created it waits for the previous thread to go the thread routine and store its thread name in another array so that it can be retained. After it has been accomplished, then SetEvent is done to signal the Thread Creation routine that it can spawn a new thread. WaitForSingleObject has been used in the Thread Creation routine to wait for the previous thread to signal that its work is over.

Application Architecture



Message Queue	Shared Global Buffer for communication between the 3 processes.
MP	Mother Process which spawns the DM process.
DM	Dynamic Manager which has 97 threads (DMG + 96 threads)
DMG	Primary thread of the DM
AM	Application Manager
AMG	Primary thread of the AM

DMG – It is the primary thread of DM process which takes care of Killing and Re-spawning DM threads as and when it receives message from MP to do so. Those DM threads which are not sending heartbeats regularly are asked to be killed by the Mother Process.

AMG – It is the AM's primary thread and it takes care of Killing and Re-spawning AM threads when it receives corresponding message from the DM. Whenever a message arrives on DM's port, it asks AMG to spawn a new thread. Similarly, whenever a thread is not sending regular heartbeats, DM sends a message to AMG to kill that thread.

Message Queue

It is a DLL, which acts as a shared global buffer for exchanging messages between the 3 processes.

- This DLL needs to share the data with its different mappings amongst the 3 processes. This is achieved by creating named data sections using #pragma statement.

#pragma bss_seg(["*section-name*"])

It specifies the default section for uninitialized data and it causes the uninitialized data allocated following the **#pragma** statement to be placed in a section specified in "section – name". In some cases, you can use **bss_seg** to speed up your load time by putting all uninitialized data in one section.

Example: To achieve this, declare the **bss_seg** section in the .DEF file as follows:

```
SECTIONS
        .bss    READ WRITE SHARED
```

and provide implementation in the .cpp file as follows:

```
#pragma bss_seg(".bss")

//data to be shared

#pragma bss_seg()
```

- Nomenclature followed for Mutexes and Events used in the message queue.

There are six message queues for the 3 processes. They are –
MP_DM – For messages sent from process DM to MP.
MP_AM – For messages sent from process AM to MP.
DM_MP – For messages sent from process MP to DM.
DM_AM – For messages sent from process AM to DM.
AM_MP – For messages sent from process MP to AM.
AM_DM – For messages sent from process DM to AM.

Mutexes created for synchronizing the writing of messages to the above message queues are as follows –

MP_DM_WriteMutex – Mutex for the queue MP_DM.
MP_AM_WriteMutex – Mutex for the queue MP_AM.
DM_MP_WriteMutex – Mutex for the queue DM_MP.
DM_AM_WriteMutex – Mutex for the queue DM_AM.
AM_MP_WriteMutex – Mutex for the queue AM_MP.
AM_DM_WriteMutex – Mutex for the queue AM_DM.

Events created for signaling the respective processes that they have message waiting for them in the message queue.

MP_Read	-	Event for the Mother Process.
DMG_Read	-	Event for DMG.
AMG_Read	-	Event for AMG.
Dxx_Read	-	Events for individual DM threads.
Axx_Read	-	Events for individual AM threads.

- The structure of the message, which is used in the message queue:

```
typedef struct msgData   //////// Message Body ////////
{
int MsgDataInt;
double MsgDataDbl;
char MsgDataChar[50];
}MsgData;

typedef enum msgtype    //////// Types of Messages /////
{
enmHeartBeat,
enmThreadKill,
enmThreadSpawn,
enmGeneralMsg
}MsgType;


typedef struct Msg          /////// Message Structure//////
{

MsgData data;
MsgType type;

} MsgStruct ;
```

Message on DM Port / HeartBeat Simulation

Heartbeats and Port-activity of the DM process have been linked to a visitor knocking on the door, and accordingly a class **CSimulateVisitor** has been made.

This class simulates a visitor standing at your door. Every now and then, you call this class to find if the visitor knocked on your door.
Sometimes, this class will respond yes and sometimes no. The probability of yes is determined by **PvisitorKnocks (PVK)**. PVK is a double, and is between 0 and 1.

If PVK is high, there is higher probability that the Visitor knocked.
A random number is generated between 0 and 1, and if this random number falls below PVK, the visitor is assumed to have knocked i.e. SUCCESS in knocking.

Also, it is possible that the visitor might stand and knock for so long that he actually sleeps. **PVisitorDies (PVD)** determines the probability that a visitor has slept off. The random-number logic for SUCCESS in going to sleep remains the same as above.
Once the visitor sleeps, he cannot knock any more until he reawakens.

There is a probability **PVisitorAwakens (PVA)** that the visitor awakens from sleep. The random-number logic for SUCCESS in reawakening remains the same as above.

In this case, every Dxx has 2 *CsimulateVisitor* objects (1 for heartbeat simulation and 1 for port-monitoring).
Every Axx has 1 *CSimulateVisitor* object (for heartbeat).

The probabilities are user definable and can be set accordingly from the *User Specification* text file.
