Thread Pooling, The Practical Way

by Primoz Gabrijelcic

Recently I was working on an application. It was not a complicated one: a simple GUI client, a server and an API to access the server. A typical small two-tier application. This time, however, the wise people in the company have decided to use SOAP for the client-server conversation (the client being the GUI app and the API layer).

It sounded pretty simple, but very soon we found SOAP requests take very different amounts of time to execute. Some requests were simple and could be executed immediately, in the thread that was running the TCP/IP (we are using ICS, so we only have one TCP thread in the server), while other requests were taking a long time (a non-indexed full text search over the embedded database managed by the same server: you get the picture) and we simply had to move them to another thread.

After talking about dynamic thread creation and thread pooling, we decided to reuse an existing mechanism: the system thread pool. That's where all the trouble started.

As we found out, the system thread pool in Windows 2000 (plus XP and 2003) is woefully inadequate for any serious use. It seems that its designer was only thinking about really trivial usage and expected everyone else to create their own thread pool. Luckily, it was possible to create a fully-fledged pooling layer based on the system thread pool and the application was saved. As I am not expecting you to take me at my word, the rest of this article will describe our solution. But first, a short introduction.

The System Thread Pool

The system thread pool was introduced in Windows 2000. It allows for a simple implementation of some frequently occurring patterns, asynchronous function execution, periodic function execution, execution when a kernel object becomes signalled, and execution when an asynchronous operation completes.

The aspect I find most interesting is the asynchronous function execution. The interface is really trivial: there is only one function, `QueueUserWorkItem`, which takes just three parameters: the address of the function to execute, an arbitrary pointer we want to pass to that function, and some flags that can slightly modify the way our function is called.

After we call `QueueUserWorkItem` the operating system will allocate a free thread from the system thread pool (a pre-allocated pool of threads) and ask it to execute our function. When the function exits, the thread will be returned to the pool and will wait for another work item to execute.

A very simple application (which is in the sub-folder `01 queue work item` in the source for this article) demonstrates the use of `QueueUserWorkItem`. As you can see in Listing 1, we simply call `QueueUserWorkItem` with the address of the function `WorkItem` and the default flags, and check the result (it should always be `True`). A moment later Windows will execute the `WorkItem` function, which will beep, sleep, and beep. Please note that this function is not called in the context of the main thread but in a context of an unknown system thread: you are advised not to call VCL functions directly and to synchronise all access to global data.

There is not much more to tell about the `QueueUserWorkItem`. If you want to read more, check the MSDN, Marcel van Braken's article *High Performance Client/Server Applications* in Issue 100, and Jeffrey Richter's excellent book *Programming Applications for Microsoft Windows, Fourth Edition*.

A (Not So Short) List of Problems

As we have seen, asynchronous execution with the system thread pool is trivial: a true 'fire and forget' operation. It does not, however, offer any control at all over the execution of the work items we are scheduling.

The first question that the documentation doesn't answer is what happens with the queued (and not yet completed) work items when an application terminates, either normally or via `TerminateProcess`.

If you were clicking rapidly on the button from the sample application I mentioned before, you may have noticed that calling `QueueUserWorkItem` doesn't guarantee that the function will start executing immediately. The operating system starts a few (typically two) functions immediately and puts the others in the queue. Only if the queue becomes too long will it start new threads and execute more functions simultaneously. The problem here is that we have no control over this process. The system thread pool will grow and shrink by itself, and we simply don't know how many work items have already started and how many are still waiting in the queue.

This brings us to another problem: if a work item is queued for a long time, maybe we would like to cancel its execution. A typical application would be the SOAP model we examined in the introduction: if the function takes too long to be executed we would simply like to return a timeout status to the client and cancel the function. The system thread pool, however, does not offer such functionality.

There is also no indication that a work item has completed its execution. We must program it by ourselves.

In simpler words: we need an infrastructure that will offer us good control over the execution of work items, a way to terminate work items (while they are running or even before they are started) and a communication mechanism that will notify us when work items are completed.

Problem Solving

Let us find out the answer to the first question: what happens to queued work items when the application terminates. The answer is quite simple: they are terminated immediately.

The proof is in the `02 cancelling queue work item` source code folder. This is a simple modification of the example from Listing 1: instead of waiting one second between the beeps, this application waits ten seconds. That gives you enough time to schedule a work item, wait for the first beep, and close the application. The second beep will never happen.

That was simple, so let's move on. How can we count the number of currently executing work items? The simplest approach is the best: we'll use a semaphore. Or better, two semaphores: one to count scheduled work items (we'll increment it when `QueueUserWorkItem` is called and decrement it at the end of the work item), another to count work items that are executing (we'll increment it at the beginning of the work item function and decrement it at the end).

The demonstration program in the source folder `03 counted work items` (a shortened version of which is shown in Listing 2) simplifies this approach some more, instead of semaphores it uses `TGpCounter`: a very thin layer around semaphores (`TGpCounter` is part of my synchronisation library `GpSync`, which is described in Issues 86 and 91).

As you can see in the code, a click on the button first increments the number of scheduled items, then calls `QueueUserWorkItem`, and in the event of a failure decrements this number. This approach, while not an example of very good coding, ensures that the counter really reflects the number of scheduled items. The `WorkItem` function first increments another counter, executes the payload (ie beep, sleep, beep) and decrements all the counters. The timer-triggered code in `Timer1Timer` checks the values of both counters and logs them on every change.

If you play with the program a little, you'll see an experimental proof for the behaviour I mentioned before. Windows will not immediately execute all the work items we are scheduling: some will wait for pool threads to become free. Only if you persistently click the button (thus growing the queue of work items waiting for free threads) will the system thread pool will be extended and new work items scheduled.

The biggest problem with this behaviour is that we have absolutely no control over work items that were queued but have not yet been allocated to a free thread. They lie in limbo and we can neither communicate with them nor terminate them. The only solution I can see is that we implement the queuing ourselves. We should keep the work items in an internal

➤ *Listing 1: Using the system thread pool for asynchronous function execution.*

```
uses
  GpWinThreadPool;
function WorkItem(context: pointer): DWORD; stdcall;
begin
  MessageBeep($FFFFFFFF);
  Sleep(1000);
  MessageBeep($FFFFFFFF);
  Result := 0; // ignored
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  if QueueUserWorkItem(@WorkItem, nil, WT_EXECUTEDEFAULT) then
    ListBox1.ItemIndex := ListBox1.Items.Add('Work item queued')
  else
    ListBox1.ItemIndex := ListBox1.Items.Add('QueueUserWorkItem failed. '+
      SysErrorMessage(GetLastError));
end;
```

➤ *Listing 2: Implementing scheduled/executing counters.*

```
const
  CCountScheduledItems =
    '/Gp/TDM/ThreadPooling/Demo/03/ScheduledWorkItems';
  CCountRunningItems =
    '/Gp/TDM/ThreadPooling/Demo/03/RunningWorkItems';
function WorkItem(context: pointer): DWORD; stdcall;
begin
  try
    TGpCounter.Increment(CCountRunningItems);
    try
      MessageBeep($FFFFFFFF);
      Sleep(1000);
      MessageBeep($FFFFFFFF);
      Result := 0; // ignored
    finally TGpCounter.Decrement(CCountRunningItems); end;
  finally TGpCounter.Decrement(CCountScheduledItems); end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  FCountScheduled.Inc;
  if QueueUserWorkItem(@WorkItem, nil, WT_EXECUTEDEFAULT)
    then
    ListBox1.ItemIndex :=
      ListBox1.Items.Add('Work item queued')
  else begin
    FCountScheduled.Dec;
    ListBox1.ItemIndex := ListBox1.Items.Add(
      'QueueUserWorkItem failed. '+
      SysErrorMessage(GetLastError));
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FCountScheduled :=
    TGpCounter.Create(CCountScheduledItems);
  FCountRunning := TGpCounter.Create(CCountRunningItems);
  FLastRunning := -1;
  FLastScheduled := -1;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeAndNil(FCountScheduled);
  FreeAndNil(FCountRunning);
end;
procedure TForm1.Timer1Timer(Sender: TObject);
var
  running  : integer;
  scheduled: integer;
begin
  running := FCountRunning.Value;
  scheduled := FCountScheduled.Value;
  if (running <> FLastRunning) or (scheduled <>
    FLastScheduled) then begin
    ListBox1.ItemIndex := ListBox1.Items.Add(Format(
      'running: %d, scheduled: %d', [running, scheduled]));
    FLastRunning := running;
    FLastScheduled := scheduled;
  end;
end;
```

queue, monitor the number of scheduled/executing work items and only call QueueUserWorkItem if not too many work items are currently executing ('not too many' being very application-dependent and as such must be set by the programmer). This approach also gives us a simple way to terminate work items not yet scheduled to the system thread pool: we just remove them from our queue.

A Little Pooling Framework

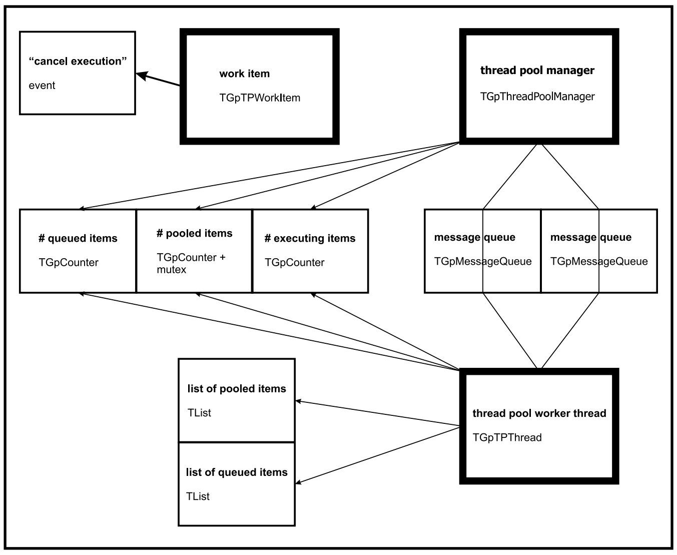
Adding a queue to the program in Listing 2 would bring us nowhere as we would be left with a totally non-reusable mess. The time has come to put some classes in the game and start thinking in an object oriented manner.

First, we need a thread pool manager (TGpThreadPoolManager), which is the central class we'll be working with. We'll set the parameters there (maximum number of queued work items, maximum time we are willing to leave a work item in the queue, and so on), use it to schedule work items and to receive a notification when work item processing will be completed.

Because we will have to store some data together with a work item, we won't be using global functions any more. Each work item will be represented by an instance of a descendent of the TGpTPWorkItem class. The scheduler will execute the virtual method Execute which we must override in our descended class.

The real workhorse is an internal thread (called TGpTPThread), which is managed by the thread pool manager. This thread accepts new work items, monitors the length of the schedule queue and the number of currently executing work items, and sends 'operation completed' notifications back to the thread pool manager. It also implements the cancellation mechanism. Communication between the thread pool manager and worker thread is implemented with two message queues (also part of the already-mentioned GpSync library).

The framework also uses three TGpCounters to communicate number of queued items (stored in the internal list), pooled items (sent to the QueueUserWorkItem but not yet executing), and executing items from worker thread back to the manager and to the application using this manager.



➤ *Figure 1*

A careful examination of Figure 1, which depicts this framework, will also show two lists managed by the thread (one holds internally queued items, another items that were already sent to the QueueUserWorkItem) and a per-work item event, which is used to cancel this event during its execution.

All this is neatly packed in the GpWinThreadPool unit.

Putting It All To Work

The code in Listing 3 (which is part of the project in this month's

➤ *Listing 3: Using the thread pooling framework.*

```
type
  TWIBeep = class(TGpTPWorkItem)
  private
    wibDelay_ms: integer;
  public
    constructor Create(delay_ms: integer);
    procedure Execute; override;
  end;
procedure TForm1.Button1Click(Sender: TObject);
var wi: TWIBeep;
begin
  wi := TWIBeep.Create(3000);
  // FManager takes the ownership of the work item
  FManager.Schedule(wi);
  Log(Format('Scheduled work item #%d', [wi.UniqueID]));
  // in all cases, status will be reported via the
  // FManager.OnWorkItemDone
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  FLastExecuting := -1;
  FLastPooled := -1;
  FLastQueued := -1;
  FManager := TGpThreadPoolManager.Create;
  FManager.MaxPooled := 3;
  FManager.MaxQueueLength := 5;
  FManager.MaxQueuedTime_sec := 3;
  FManager.OnWorkItemDone := HandleWorkItemDone;
  FManager.OnThreadError := HandleThreadError;
end;
procedure TForm1.HandleWorkItemDone(sender: TObject;
  workItem: TGpTPWorkItem);
var msg: string;
begin
  case workItem.Status of
    tpmsCompleted : msg := 'Completed.';
    tpmsThreadTimeout  : msg := 'Timed out waiting on thread
      queue.';
    tpmsThreadQueueFull: msg := 'Thread queue is full.';
    tpmsException      : msg := 'Work item raised an
      exception.';
    tpmsWin32Error     : msg := 'QueueUserWorkItem failed.';
    tpmsCanceled       : msg := 'Work item was canceled.';
    tpmsServerBusy     : msg := 'Server is busy.';
    else                 msg := 'Unknown error has occured.';
  end; //case
  Log(Format('#%d: %s %s', [(workItem as TWIBeep).UniqueID,
    msg, workItem.LastError]));
end; { TForm1.HandleWorkItemDone }
{ TWIBeep }
constructor TWIBeep.Create(delay_ms: integer);
begin
  wibDelay_ms := delay_ms;
end;
procedure TWIBeep.Execute;
var
  iRound     : integer;
  sleepRounds: integer;
begin
  MessageBeep($FFFFFFFF);
  sleepRounds := wibDelay_ms div 100;
  for iRound := 1 to sleepRounds do begin
    Sleep(100);
    if Canceled then Exit;
  end; //for
  MessageBeep($FFFFFFFF);
end;
```

source called 04 manager and work item classes) demonstrates how the framework is used. Some parts were removed for brevity: check the source for the full story.

As you can see, the WorkItem function was replaced with the TWIBeep class. The method Button1Click simply creates a new instance of this class and passes it to the FManager's Schedule method. It then prints the unique ID that was given to this work item and exits.

FManager is an instance of the TGpThreadPoolManager class created in the FormCreate handler. The code sets the parameters to a maximum of three work items sent to the QueueUserWorkItem (MaxPooled) at any time, a maximum of five items in the internal queue (MaxQueueLength), and a maximum work item lifetime of three seconds (MaxQueuedTime_sec). It also sets an error handler (which should never be called if my code is perfect, as it only reports internal errors) and the work item completed handler.

After the manager does its magic, the same work item instance is passed to the OnWorkItemDone handler (method HandleWorkItemDone). It checks the status of the work item and prints it out. If an error occurred, it will also print error description.

A point worth mentioning is that the OnWorkItemDone will always be called, even if the thread blocks and stops receiving messages from the thread manager, if QueueUserWorkItem fails, or if the work item's Execute method causes an exception. Therefore, you should always check the work item's Status in the OnWorkItemDone handler.

In the end TWIBeep.Execute received a facelift too. It does its sleeping in small steps (100ms each) and after each step checks to see if it has been cancelled (if the application executed FManager.CancelAll). If that happens, Execute will exit immediately.

Work Item Flow

The best way to understand the innards of the GpWinThreadPool unit is to follow a work item through the framework. As we have already seen, the work item is first passed to the thread manager's Schedule function. It assigns a new unique ID to the work item and marks the time it was received. This time is stored in UTC as we really don't want a change to Daylight Saving Time during the work item processing to break our calculations (see Issue 65 for more information on time calculations). The work item is then sent to the worker thread via the message queue.

The worker thread's Schedule method sets some internal parameters the work item will need later. Next it checks if the internal queue is already too long, in which case the work item will be immediately refused (it will be sent back to the thread manager via the message queue and the thread manager will call the OnWorkItemDone handler). If the queue is not too long, the work item will be added to the internal queue and Schedule will trigger another part of the scheduler by signalling an event.

This event will cause ScheduleNext to be called. If there are not too many outstanding work items already being processed by the QueueWorkUserItem, it will call that function (via the internal method ScheduleWorkItem). After that, it will check if any work item in the internal queue is already too old, in which case it will be refused with the 'server busy' status.

ScheduleWorkItem calls QueueWorkUserItem, always passing it the same function, TPWorkItemExecutor, but setting the context parameter to the work item instance.

Now the story stalls for an indefinite time. At some moment later our friendly Operating System starts executing the TPWorkItemExecutor. This is a simple function which maps the context parameter back to the work item instance and calls its Schedule method.

Schedule does the work in a similar way to the WorkItem in Listing 2: it increments and decrements counters, and calls the work item's Execute somewhere in between. Besides that, it traps exceptions and maps them to the work item's Status and LastError properties. Importantly, it checks at the very beginning if the work item was already cancelled and exits immediately if this is true.

When Schedule finishes its work, TPWorkItemExecutor calls the method assigned to the work item's wiOnDone handler. In our case, this is HandleOnDone method (from the worker thread).

HandleOnDone (which is called from the context of the system thread, not the thread pool worker thread!) synchronises the completion notification by sending a message and an address of the work item instance back to the worker thread via the same message queue that is used for manager-to-thread communication.

The worker thread processes this message in the RequestCompleted method. It removes the work item from the internal queues, sends the work item instance to the thread manager, and sets the internal scheduler event, which will cause the ScheduleNext to be called again and possibly schedule a new work item.

Finally, the thread manager processes this message in its own RequestCompleted method, which simply calls the OnWorkItemDone handler (if assigned) and destroys the work item request.

That is the end of a work item's long journey. I don't have much to add to that, except maybe that if you intend to use the TGpThreadPoolManager from a thread, you should make sure that somebody in this thread will process Windows messages, because the message queue used to send the confirmation messages from the worker thread back to the manager depends on that. The easiest way of achieving that is to use MsgWaitForMultipleObjects(.., QS_ALLINPUT) instead of a simple WaitFor... in the thread's main loop.

Primoz Gabrijelcic is the R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com