



Getting Full Speed with Delphi

[Why Single-Threading Is Not Enough?]

Primož Gabrijelčič
primoz@gabrijelcic.org

The Free Lunch is Over

For the last fifty years, we programmers had it easy. We could write slow, messy, suboptimal code and when a customer complained we would just say: "No problem, with the next year computers the software will be quick as a lightning!" With some luck new hardware would solve the problem and if not we could pretend to fix the problem until new generation of computers came out. In other words - Moore's law worked in our favor.

This situation changed radically in the past few years. This situation changed radically in the last year. New processors are not significantly faster than the old ones and unless something will drastically change in CPU design and production, that will stay so. Instead of packing more speed, manufacturers are now putting multiple processor units (or *cores* as they are usually called) inside one CPU. In a way that gives our customers faster computers, but only if they are using multiple programs at once. Our traditionally written programs that can use only one processor unit at any moment won't profit from multiple cores.

As we can all see, this is not good for us, programmers. We have to do something to make our programs faster on multi-core processors. The only way to do that is to make the program do more than one thing at the same time and the simplest and most effective way to do it is to use *multithreading* or using the ability of the operating system to execute multiple *threads* simultaneously. [A note to experienced readers: There's more to threads, threading and multithreading than I will tell in today's presentation. If you want to get a full story, check the Wikipedia, [en.wikipedia.org/wiki/Thread_\(computer_science\)](http://en.wikipedia.org/wiki/Thread_(computer_science)).]

Multithreading

As a programmer you probably know, at least instinctively, what is a *process*. In operating system terminology, a process is a rough equivalent of an application - when the user starts an application, operating system creates and starts new process. Process contains (or better, owns) application code, but also all resources that this code uses - memory, file handles, device handles, sockets, windows etc.


When the program is executing, the system must also keep track of the current execution address, state of the CPU registers and state of the program's stack. This information, however, is not part of the process, but belongs to a *thread*. Even a simplest program uses one thread, which describes the program's execution. In other words, process encapsulates program's static data while thread encapsulates the dynamic part. During the program's lifetime, the thread describes its line of execution - if we know the state of the thread at every moment, we can fully reconstruct the execution in all details.

All operating systems support one thread per process (obviously) but some go further and support multiple threads in one process. Actually, most modern operating systems support *multithreading* (as this approach is called), the difference is just in details. With multithreading, operating system manages multiple execution paths through the same code and those paths may execute at the same time (and then again, they may not - but more on that later).

An important fact is that processes are *heavy*. It takes a long time (at least at the operating system level where everything is measured in microseconds) to create and load a new process. In contrast to that, *threads* are light. New thread can be created almost immediately - all the operating system has to do is to allocate some memory for the stack and set up some control structures used by the kernel.

Another important point about processes is that they are isolated. Operating system does its best to separate one process from another so that buggy (or malicious) code in one process cannot crash another process (or read private data from it). If you're old enough to remember Windows 3 where this was not the case you can surely appreciate the stability this isolation is bringing to the user. In contrast to that, multiple threads inside a process share all process resources - memory, file handles and so on. Because of that, threading is inherently fragile - it is very simple to bring down one thread with a bug in another.

In the beginning, operating systems were single-tasking. In other words, only one task (i.e. process) could be executing at the same time



and only when it completed the job (when the task terminated), new task can be scheduled (started).

As soon as the hardware was fast enough, multitasking was invented. Most computers still had only one but through the operating system magic it looked like this processor is executing multiple programs at the same time. Each program was given a small amount of time to do its job after which it was paused and another program took its place. After some indeterminate time (depending on the system load, number of higher priority tasks etc.) the program could execute again and operating system would run it from the position in which it was paused, again only for the small amount of time. In technical terms, processor registers were loaded from some operating system storage immediately before the program was given its time to run and were stored back to this storage when program was paused.

Two very different approaches to multitasking are in use. In *cooperative* multitasking, the process itself tells the operating system when it is ready to be paused. This simplifies the operating system but gives a badly written program an opportunity to bring down whole computer. Remember Windows 3? That was cooperative multitasking at its worst.

Better approach is *pre-emptive* multitasking where each process is given its allotted time (typically about 55 milliseconds on a PC) and is then pre-empted; that is, hardware timer fires and takes control from the process and gives it back to the operating system which can then schedule next process. This approach is used in Windows 95, NT and all their successors. That way, multitasking system can appear to execute multiple processes at once event if it has only one processor core. Things go even better if there are multiple cores inside the computer as multiple processes can really execute at the same time then.

The same goes for threads. Single-tasking systems were limited to one thread per process by default. Some multitasking systems were single-threaded (i.e. they could only execute one thread per process) but all modern Windows are multithreaded - they can execute multiple threads inside one process. Everything I said about multitasking applies to threads too. Actually, it is the threads that are scheduled, not processes.

Problems and Solutions

Multithreading can bring you speed, but it can also bring you grey hair. There are many possible problems which you can encounter in multithreaded code that will never appear in a single-threaded program.

For example, splitting task into multiple threads can make the execution slower instead of faster. There are not many problems that can be nicely parallelized and in most cases we must pass some data from one thread to another. If there's too much communication between threads, communication can use more CPU than the actual, data processing code.

Then there's a problem of data sharing. When threads share data, we must be very careful to keep this data in a consistent state. For example, if two threads are updating shared data, it may end in a mixed state where half the data was written by the first thread and another half by the second.

This problem, *race condition* as it's called, is usually solved by some kind of *synchronization*. We use some kind of locking (critical sections, mutexes, spinlocks, semaphores) to make sure that only one thread at a time can update the data. However, that brings us another problem or two. Firstly, synchronization makes the code slower. If two threads try to enter such locked code, only one will succeed and another will be temporarily suspended and our clever, multithreaded program will again use only one CPU core.

Secondly, synchronization can cause *deadlocks*. This is a state where two (or more) threads forever wait on each other. For example, thread A is waiting on a resource locked by thread B and thread B is waiting on a resource locked by thread A. Not good. Deadlocks can be very tricky; easy to introduce into the code and hard to find.

There's a way around synchronization problems too. You can avoid data sharing and use messaging systems to pass data around or you can use well-tested lock-free structures for data sharing. That doesn't solve the problem of *livelocks* though. In livelock state, two (or more) threads are waiting on some resource that will never be freed because the other thread is using it, but they do that dynamically - they're not waiting for some synchronization object to become released. The code is executing and threads are alive, they can just not enter a state where all conditions will be satisfied at once.

Four Paths to Multithreading

There's more than one way to skin a cat (supposedly) and there's more than one way to create a thread. Of all the options I have selected four more interesting to the Delphi programmer.

The Delphi Way

Creating a thread in Delphi is as simple as declaring a class that descends from the TThread class (which lives in the Classes unit), overriding its Execute method and instantiating an object of this class (in other words, calling TMyThread.Create). Sounds simple, but the devil is, as always, in the details.

```
TMyThread = class(TThread)
protected
  procedure Execute; override;
end;

FThread1 := TMyThread1.Create;
```

The Windows Way

Surely, the TThread class is not complicated to use but the eternal hacker in all of us wants to know – how? How is TThread implemented? How do threads function at the lowest level. It turns out that the Windows' threading API is not overly complicated and that it can be easily used from Delphi applications.

It's easy to find the appropriate API, just look at the TThread.Create. Besides other things it includes the following code (Delphi 2007):

```
FHandle := BeginThread(nil, 0, @ThreadProc, Pointer(Self),
  CREATE_SUSPENDED, FThreadId);
if FHandle = 0 then
  raise EThread.CreateResFmt(@SThreadCreateError,
    [SysErrorMessage(GetLastError)]);
```

If we follow this a level deeper, into BeginThread, we can see that it calls CreateThread. A short search points out that this is a Win32 kernel function, and a look into the MSDN confirms that it is indeed a true and proper way to start a new thread.

One thing has to be said about the Win32 threads – why to use them at all? Why go down to the Win32 API if the Delphi's TThread is so more comfortable to use? I can think of two possible answers.

Firstly, you would use Win32 threads if working on a multi-language application (built using DLLs compiled with different compilers) where

threads objects are passed from one part to another. A rare occasion, I'm sure, but it can happen.

Secondly, you may be creating lots and lots of threads. Although that is not really something that should be recommended, you may have a legitimate reason to do it. As the Delphi's TThread uses 1 MB of stack space for each thread, you can never create more than (approximately) 2000 threads. Using CreateThread you can provide threads with smaller stack and thusly create more threads – or create a program that successfully runs in a memory-tight environment. If you're going that way, be sure to read great blog post by Raymond Chen at blogs.msdn.com/oldnewthing/archive/2005/07/29/444912.aspx.

The Lightweight Way


From complicated to simple ... There are many people on the Internet who thought that Delphi's approach to threading is overly complicated (from the programmer's viewpoint, that it). Of those, there are some that decided to do something about it. Some wrote components that wrap around TThread, some wrote threading libraries, but there's also a guy that tries to make threading as simple as possible. His name is Andreas Hausladen (aka Andy) and his library (actually it's just one unit) is called AsyncCalls and can be found at andy.jgknet.de/blog/?page%5Fid=100.

AsyncCalls is very generic as it supports all Delphis from version 5 onwards. It is licensed under the Mozilla Public License 1.1, which doesn't limit the use of AsyncCalls inside commercial applications. The only downside is that the documentation is scant and it may not be entirely trivial to start using AsyncCalls for your own threaded code. Still, there are some examples on the page linked above. This article should also help you started.

To create and start a thread (there is no support for creating threads in suspended state), just call AsyncCall method and pass it the name of the main thread method.

```
procedure TfrmTestAsyncCalls.btnStartThread1Click(Sender: TObject);  
begin  
    FThreadCall11 := AsyncCall(ThreadProc1, integer(@FStopThread1));  
    Log('Started thread'); // AsyncCalls threads have no IDs  
end;
```

AsyncCalls is a great solution to many threading problems. As it is actively developed, I can only recommend it.



The No-Fuss Way

I could say that I left the best for the end but that would be bragging. Namely, the last solution I'll describe is of my own making.

OmniThreadLibrary (OTL for short) approaches the threading problem from a different perspective. The main design guideline was: "Enable the programmer to work with threads in as fluent way as possible." The code should ideally relieve you from all burdens commonly associated with multithreading. I'm the first to admit that the goal was not reached yet, but I'm slowly getting there.

The bad thing is that OTL has to be learned. It is not a simple unit that can be grasped in an afternoon, but a large framework with lots of functions. On the good side, there are many examples (otl.17slon.com/tutorials.htm; you'll also find download links there). On the bad side, the documentation is scant. Sorry for that, but you know how it goes – it is always more satisfying to program than to write documentation. Another downside is that it supports only Delphi 2007 and newer. OTL is released under the BSD license which doesn't limit you from using it in commercial applications in any way.

OTL is a message based framework and uses custom, extremely fast messaging system. You can still use any blocking stuff and write TThread-like multithreading code, if you like. Synchronize is, however, not supported. Why? Because I think it's a bad idea, that's why.

While you can continue to use low-level approach to multithreading, OTL supports something much better – high-level primitives.

High Level Multithreading

At this moment (March 2011), OmniThreadLibrary supports five high-level multithreading concepts:

- Join
- Future
- Pipeline
- Fork/Join
- Parallel for

The implementation of those tools actively uses anonymous methods which is why they are supported only in Delphi 2009 and newer.

Those tools help the programmer to implement multithreaded solution without thinking about thread creation and destruction. All those tools are implemented in the OtlParallel unit.

Join

The simplest of those tools is Join. It allows you to start multiple background tasks and wait until they have all completed. No result is returned – at least directly, as you can always store result into a shared variable. If your code returns a result, a better approach may be to use a Future or Fork/Join.

A simple demonstration of Join (below) starts two tasks – one sleeps for two and another for three seconds. When you run this code, Parallel.Join will create two background threads and run RunTask1 in first and RunTask2 in second. It will then wait for both threads to complete their work and only then the execution of main thread will continue.

```
procedure TfrmOTLDemoJoin.btnParallelClick(Sender: TObject);  
begin  
    Parallel.Join([RunTask1, RunTask2]);  
end;
```

```
procedure TfrmOTLDemoJoin.RunTask1;  
begin  
    Sleep(2000);  
end;
```

```
procedure TfrmOTLDemoJoin.RunTask2;  
begin  
    Sleep(3000);  
end;
```


Join takes special care for compatibility with single-core computers. If you run the above code on a single-core machine (or if you simply limit the process to one core), it will simply execute tasks sequentially, without creating a thread.

Join accepts anonymous methods. The above demo could also be coded as a single method executing two anonymous methods.

```
procedure TfrmOTLDemoJoin.btnAnonymousClick(Sender: TObject);  
begin  
    Parallel.Join(  
        procedure begin  
            Sleep(2000);  
        end,  
        procedure begin  
            Sleep(3000);  
        end);  
end;
```

There are four overloaded Join methods. Two are accepting two tasks and two are accepting any number of tasks. [The first demo above uses latter version of Join and the second demo the former version.]

Two version of Join accept *procedure (task: IOmniTask)* instead of a simple *procedure* and can be used if you have to communicate with the main thread during the execution. To do so, you would have to learn more about communication and tasks, which will be covered later in this document.

```
class procedure Join(const task1, task2: TProc); overload;  
class procedure Join(const task1, task2: TOmniTaskDelegate); overload;  
class procedure Join(const tasks: array of TProc); overload;  
class procedure Join(const tasks: array of TOmniTaskDelegate); overload;
```

Join is demonstrated in demo 37_ParallelJoin (part of the OmniThreadLibrary package).

Future

“They (futures) describe an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed.”

– Wikipedia

Futures are a tool that help you start background calculation and then forget about it until you need the result of the calculation.

To start background calculation, you simply create a IOmniFuture instance of a specific *type* (indicating the type returned from the calculation).

```
Future := Parallel.Future<type>(calculation);
```

Calculation will start in background and main thread can continue with its work. When the calculation result is needed, simply query `Future.Value`. If the calculation has already completed its work, value will be returned immediately. If not, the main thread will block until the background calculation is done.

The example below starts background calculation that calculates number of prime numbers in interval 1..1000000. While the calculation is running, it uses main thread for “creative” work – outputting numbers into listbox and sleeping. At the end, calculation result is returned by querying `future.Value`.

```
procedure TfrmOTLDemoFuture.btnCalcFutureClick(Sender: TObject);  
const  
    CMaxPrimeBound = 1000000;  
var  
    future    : IOmniFuture<integer>;  
    i        : integer;  
    numPrimes: integer;  
begin  
    future := Parallel.Future<integer>(  
        function: integer  
        begin  
            Result := CountPrimesTo(CMaxPrimeBound);  
        end  
    );  
  
    for i := 1 to 10 do begin  
        lbLog.Items.Add(IntToStr(i));  
        Sleep(20);  
        lbLog.Update;  
    end;  
  
    Log(Format('Num primes up to %d: %d', [CMaxPrimeBound, future.Value]));  
end;
```

As with `Join`, there are two `Future<T>` overloads, one exposing the internal *task* parameter and another not.

```
class function Future<T>(action: TOmniFutureDelegate<T>): IOmniFuture<T>; overload;  
class function Future<T>(action: TOmniFutureDelegateEx<T>): IOmniFuture<T>; overload;
```

`IOmniFuture<T>` has some other useful features. You can cancel the calculation (*Cancel*) and check if calculation has been cancelled (*IsCancelled*). You can also check if calculation has already completed (*IsDone* and *TryValue*).

```
IOmniFuture<T> = interface  
    procedure Cancel;
```

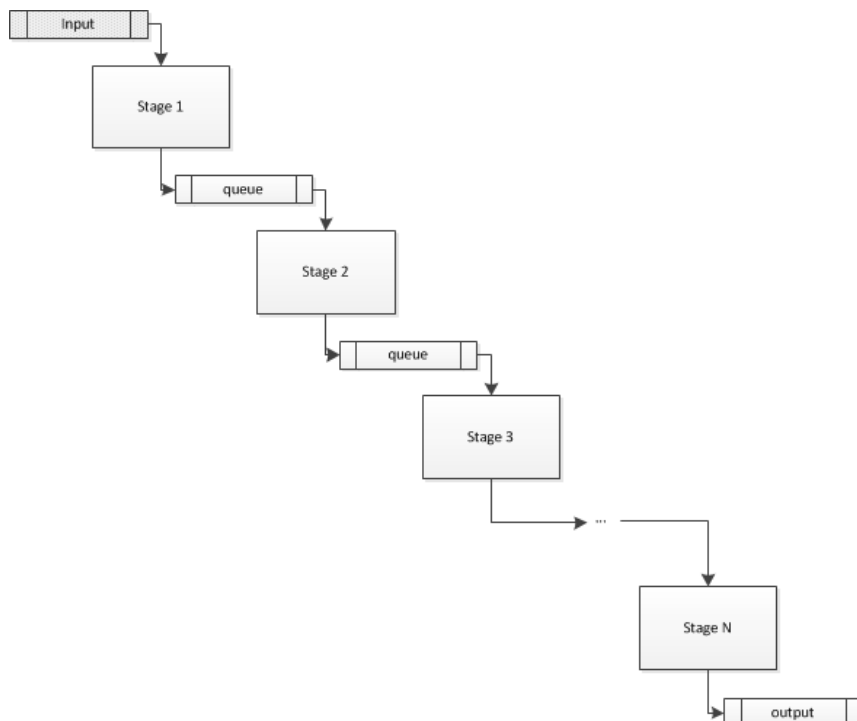
```
function IsCancelled: boolean;  
function IsDone: boolean;  
function TryValue(timeout_ms: cardinal; var value: T): boolean;  
function Value: T;  
end;
```

Futures are demoed in project 39_Futures. They were also topic of my blog post www.thedelphigeek.com/2010/06/omnithreadlibrary-20-sneak-preview-2.html.

Interestingly, futures can be very simply implemented on top of Delphi's TThread. I wrote about that in www.thedelphigeek.com/2010/06/future-of-delphi.html.

Pipeline

Pipeline construct implements high-level support for multistage processes. The assumption is that the process can be split into stages (or subprocesses), connected with data queues. Data flows from the (optional) input queue into the first stage, where it is partially processed and then emitted into intermediary queue. First stage then continues execution, processes more input data and outputs more output data. This continues until complete input is processed. Intermediary queue leads into the next stage which does the processing in a similar manner and so on and on. At the end, the data is output into a queue which can be then read and processed by the program that created this multistage process. As a whole, a multistage process functions as a pipeline – data comes in, data comes out.

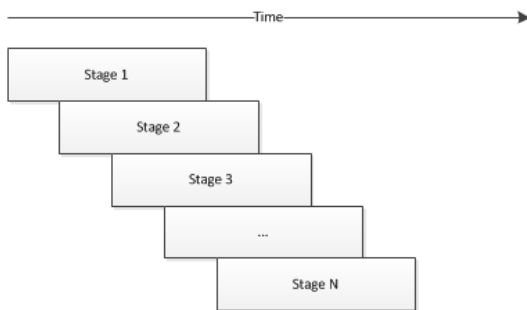


What is important here is that no stage shares state with any other stage. The only interaction between stages is done with the data passed through the intermediary queues. The quantity of data, however, doesn't have to be constant. It is entirely possible for a stage to generate more or less data than it received on input.

In a classical single-threaded program the execution plan for a multistage process is very simple.



In a multithreaded environment, however, we can do better than that. Because the stages are largely independent, they can be executed in parallel.



A pipeline is created by calling `Parallel.Pipeline` function which returns `IOmniPipeline` interface. There are two overloaded versions – one for general pipeline building and another for simple pipelines that don't require any special configuration.

```
class function Pipeline: IOmniPipeline; overload;  
class function Pipeline(  
    const stages: array of TPipelineStageDelegate;  
    const input: IOmniBlockingCollection = nil):  
    IOmniPipeline; overload;
```

The latter version takes two parameters – an array of processing stages and an optional input queue. Input queue can be used to provide initial data to the first stage. It is also completely valid to pass 'nil' for the input queue parameter and run the first stage without any input.

Blocking collections (they are covered later in this document) are used for data queuing in the `Parallel.Pipeline` implementation.

Stages are implemented as anonymous procedures, procedures or methods taking two queue parameters – one for input and one for output. Except in the first stage where the input queue may not be defined, both are automatically created by the Pipeline implementation and passed to the stage delegate.

```
TPipelineStageDelegate = reference to procedure  
  (const input, output: IOmniBlockingCollection);
```

The next code fragment shows a simple pipeline containing five stages. Result of Parallel.Pipeline is a IOmniBlockingCollection, which is a kind of single-ended queue. Result is accessed by reading an element from this queue (by calling pipeOut.Next), which will block until this element is ready.

```
procedure TfrmOTLDemoPipeline.btnCalcPipelineClick(Sender: TObject);  
var  
  pipeOut: IOmniBlockingCollection;  
begin  
  pipeOut := Parallel.Pipeline([  
    StageGenerate,  
    StageMult2,  
    StageMinus3,  
    StageMod5,  
    StageSum]  
  ).Run;  
  
  Log(Format('Pipeline result: %d', [pipeOut.Next.AsInteger]));  
end;
```

Pipeline stages are shown below. First stage ignores the input (which is not provided) and generates elements internally. Each element is written to the output queue.

```
procedure StageGenerate(const input, output: IOmniBlockingCollection);  
var  
  i: integer;  
begin  
  for i := 1 to CNumTestElements do  
    if not output.TryAdd(i) then Exit;  
end;
```

Next three stages are reading data from input (by using for..in loop), and outputting modified data into output queue. For..in will automatically terminate when previous stage terminates and input queue runs out of data.

```
procedure StageMult2(const input, output: IOmniBlockingCollection);  
var  
  value: TOmniValue;  
begin  
  for value in input do
```

```

        if not output.TryAdd(2 * value.AsInteger) then
            Exit;
        end;

procedure StageMinus3(const input, output: IOmniBlockingCollection);
var
    value: TOmniValue;
begin
    for value in input do
        if not output.TryAdd(value.AsInteger - 3) then
            Exit;
        end;
end;

procedure StageMod5(const input, output: IOmniBlockingCollection);
var
    value: TOmniValue;
begin
    for value in input do
        if not output.TryAdd(value.AsInteger mod 5) then
            Exit;
        end;
end;

```

The last stage also reads data from input but outputs only one number – a sum of all input values.

```

procedure StageSum(const input, output: IOmniBlockingCollection);
var
    sum : integer;
    value: TOmniValue;
begin
    sum := 0;
    for value in input do
        Inc(sum, value);
        output.TryAdd(sum);
    end;


```

The full power of the IOmniPipeline interface is usually accessed via the parameterless Parallel.Pipeline function.

```

IOmniPipeline = interface
    procedure Cancel;
    function Input(const queue: IOmniBlockingCollection): IOmniPipeline;
    function NumTasks(numTasks: integer): IOmniPipeline;
    function Run: IOmniBlockingCollection;
    function Stage(pipelineStage: TPipelineStageDelegate): IOmniPipeline; overload;
    function Stage(pipelineStage: TPipelineStageDelegateEx): IOmniPipeline;
        overload;
    function Stages(const pipelineStages: array of TPipelineStageDelegate):
        IOmniPipeline; overload;
    function Stages(const pipelineStages: array of TPipelineStageDelegateEx):
        IOmniPipeline; overload;
    function Throttle(numEntries: integer; unblockAtCount: integer = 0):
        IOmniPipeline;
end;

```



Input sets the input queue. If it is not called, input queue will not be assigned and the first stage will receive nil for the input parameter.

Stage adds one pipeline stage.

Stages adds multiple pipeline stages.

NumTasks sets the number of parallel execution tasks for the stage(s) just added with the *Stage(s)* function (IOW, call *Stage* followed by *NumTasks* to do that). If it is called before any stage is added, it will specify the default for all stages. Number of parallel execution tasks for a specific stage can then still be overridden by calling *NumTasks* after the *Stage* is called.

Throttle sets the throttling parameters for stage(s) just added with the *Stage(s)* function. Just as the *NumTask* it affects either the global defaults or just currently added stage(s). By default, throttling is set to 10240 elements.

Run does all the hard work – creates queues and sets up OmniThreadLibrary tasks. It returns the output queue which can be then used in your program to receive the result of the computation. Even if the last stage doesn't produce any result this queue can be used to signal the end of computation.

Read more about pipelines in the OmniThreadLibrary on www.thedelphigeek.com/2010/11/multistage-processes-with.html.

Pipelines are demoed in project 41_Pipelines.

Fork/Join

Fork/Join is an implementation of “Divide and conquer” technique. In short, Fork/Join allows you to:

- Execute multiple tasks
- Wait for them to terminate
- Collect results

The trick here is that subtasks may spawn new subtasks and so on ad infinitum (probably a little less, or you're run out of stack ;)). For optimum execution, Fork/Join must there for guarantee that the code is never running too much background threads (an optimal value is usually equal to the number of cores in the system) and that those threads don't run out of work.

Fork/Join subtasks are in many way similar to Futures. They offer slightly less functionality (no cancellation support) but they are enhanced in another way – when Fork/Join subtasks runs out of work, it will start executing some other task's workload keeping the system busy.

A typical way to use Fork/Join is to create an IOmniForkJoin<T> instance

```
forkJoin := Parallel.ForkJoin<integer>;
```

and then create computations owned by this instance

```
max1 := forkJoin.Compute(  
  function: integer begin  
    Result := ...  
  end);  
max2 := forkJoin.Compute(  
  function: integer begin  
    Result := ...  
  end);
```

To access computation result, simply call computation object's Value function.

```
Result := Max(max1.Value, max2.Value);
```

The code below shows how Fork/Join can be used to find maximum element in an array. At each computation level, ParallelMaxRange receives a slice of original array. If it is small enough, sequential function is called to determine maximum element in the slice. Otherwise, two subcomputations are created, each working on one half of the original slice.

```
function ParallelMaxRange(const forkJoin: IOmniForkJoin<integer>;  
  intarr: PIntArray; low, high, cutoff: integer): integer;  
  
function Compute(low, high: integer): IOmniCompute<integer>;  
begin  
  Result := forkJoin.Compute(  
    function: integer  
    begin  
      Result := ParallelMaxRange(forkJoin, intarr, low, high, cutoff);  
    end  
  );  
end;  
  
var  
  max1: IOmniCompute<integer>;  
  max2: IOmniCompute<integer>;  
  mid : integer;  
begin  
  if (high-low) < cutoff then
```

```

    Result := SequentialMaxRange(intarr, low, high)
else begin
    mid := (high + low) div 2;
    max1 := Compute(low, mid);
    max2 := Compute(mid+1, high);
    Result := Max(max1.Value, max2.Value);
end;
end;

function TfrmOTLDemoForkJoin.RunParallel(intarr: PIntArray; low, high,
    cutoff: integer): integer;
begin
    Result := ParallelMaxRange(Parallel.ForkJoin<integer>, intarr, low, high, cutoff);
end;

```

As this is a very recent addition to OmniThreadLibrary (presented first time here at ADUG), there are no demos or blog articles that would help you understand the Fork/Join. Stay tuned!

Parallel For

Parallel For (actually called ForEach because For would clash with the reserved keyword **for**) is a construct that enumerates in a parallel fashion over different containers. The most typical usage is enumerating over range of integers (just like in the classical **for**), but it can also be used similar to the **for.in** – for enumerating over Delphi- or Windows-provided wnumerators.

A very simple example loops over an integer range and increments a global counter for each number that is also a prime number. In other way, the code below counts number of primes in range 1..CHighPrimeBound.

```

procedure TfrmOTLDemoParallelFor.btnParallelClick(Sender: TObject);
var
    numPrimes: TGp4AlignedInt;
begin
    numPrimes.Value := 0;
    Parallel
        .ForEach(2, CHighPrimeBound)
        .Execute(
            procedure (const value: integer)
            begin
                if IsPrime(value) then
                    numPrimes.Increment;
            end
        );

    Log(Format('%d primes', [numPrimes.Value]));
end;

```

If you have data in a container that supports enumeration (with one limitation – enumerator must be implemented as a class, not as an interface or a record) then you can enumerate over it in parallel.

```
nodeList := TList.Create;  
// ...  
Parallel.ForEach<integer>(nodeList).Execute(  
    procedure (const elem: integer)  
    begin  
        if IsPrime(elem) then  
            outQueue.Add(elem);  
    end);
```

[Note: The outQueue parameter is of type IOmniBlockingCollection which allows Add to be called from multiple threads simultaneously.]

ForEach backend allows parallel loops to be executed asynchronously. In the code below, parallel loop tests numbers for primeness and adds primes to a TOmniBlockingCollection queue. A normal for loop, executing in parallel with the parallel loop, reads numbers from this queue and displays them on the screen.

```
var  
    prime      : TOmniValue;  
    primeQueue: IOmniBlockingCollection;  
begin  
    lbLog.Clear;  
    primeQueue := TOmniBlockingCollection.Create;  
  
    Parallel.ForEach(1, 1000).NoWait  
        .OnStop(  
            procedure  
            begin  
                primeQueue.CompleteAdding;  
            end)  
        .Execute(  
            procedure (const value: integer)  
            begin  
                if IsPrime(value) then begin  
                    primeQueue.Add(value);  
                end;  
            end);  
  
    for prime in primeQueue do begin  
        lbLog.Items.Add(IntToStr(prime));  
        lbLog.Update;  
    end;  
end;
```

This code depends on a TOmniBlockingCollection feature, namely that the enumerator will block when the queue is empty unless CompleteAdding is called. That's why the OnStop delegate must be

provided – without it the “normal” for loop would never stop. (It would just wait forever on the next element.)

While this shows two powerful functions (NoWait and OnStop) it is also kind of complicated and definitely not a code I would want to write too many times. That’s why OmniThreadLibrary also provides a syntactic sugar in a way of the Into function.

```
var
    prime      : TOmniValue;
    primeQueue: IOmniBlockingCollection;
begin
    lbLog.Clear;
    primeQueue := TOmniBlockingCollection.Create;

    Parallel.ForEach(1, 1000).PreserveOrder.NoWait
        .Into(primeQueue)
        .Execute(
            procedure (const value: integer; var res: TOmniValue)
            begin
                if IsPrime(value) then
                    res := value;
            end);
    for prime in primeQueue do begin
        lbLog.Items.Add(IntToStr(prime));
        lbLog.Update;
    end;
end;
```

This code demos few different enhancements to the ForEach loop. Firstly, you can order the Parallel subsystem to preserve input order by calling the PreservedOrder function. Secondly, because Into is called, ForEach will automatically call CompleteAdding on the parameter passed to the Into when the loop completes. No need for the ugly OnStop call.

Thirdly, Execute (also because of the Into) takes a delegate with a different signature. Instead of a standard ForEach signature *procedure (const value: T)* you have to provide it with a *procedure (const value: integer; var res: TOmniValue)*. If the output parameter (res) is set to any value inside this delegate, it will be added to the Into queue and if it is not modified inside the delegate, it will not be added to the Into queue.

If you want to iterate over something very nonstandard, you can write a “GetNext” delegate (parameter to the ForEach<T> itself):

```

Parallel.ForEach<integer>(
    function (var value: integer): boolean
    begin
        value := i;
        Result := (i <= testSize);
        Inc(i);
    end)
.Execute(
    procedure (const elem: integer)
    begin
        outQueue.Add(elem);
    end);

```

In case you wonder what the possible iteration sources are, here's the full list:

```

ForEach(const enumerable: IOmniValueEnumerable): IOmniParallelLoop;
ForEach(const enum: IOmniValueEnumerator): IOmniParallelLoop;
ForEach(const enumerable: IEnumerable): IOmniParallelLoop;
ForEach(const enum: IEnumerator): IOmniParallelLoop;
ForEach(const sourceProvider: TOmniSourceProvider): IOmniParallelLoop;
ForEach(enumerator: TEnumeratorDelegate): IOmniParallelLoop;
ForEach(low, high: integer; step: integer = 1): IOmniParallelLoop<integer>;
ForEach<T>(const enumerable: IOmniValueEnumerable): IOmniParallelLoop<T>;
ForEach<T>(const enum: IOmniValueEnumerator): IOmniParallelLoop<T>;
ForEach<T>(const enumerable: IEnumerable): IOmniParallelLoop<T>;
ForEach<T>(const enum: IEnumerator): IOmniParallelLoop<T>;
ForEach<T>(const enumerable: TEnumerable<T>): IOmniParallelLoop<T>;
ForEach<T>(const enum: TEnumerator<T>): IOmniParallelLoop<T>;
ForEach<T>(enumerator: TEnumeratorDelegate<T>): IOmniParallelLoop<T>;
ForEach(const enumerable: TObject): IOmniParallelLoop;
ForEach<T>(const enumerable: TObject): IOmniParallelLoop<T>;

```

The last two versions are used to iterate over any object that supports class-based enumerators. Sadly, this feature is only available in Delphi 2010 because it uses extended RTTI to access the enumerator and its methods.

A special care has been taken to achieve fast execution. Worker threads are not fighting for input values but are cooperating and fetching input values in blocks.

The backend allows for efficient parallel enumeration even when the enumeration source is not threadsafe. You can be assured that the data passed to the ForEach will be accessed only from one thread at the same time (although this will not always be the same thread). Only in special occasions, when backend knows that the source is threadsafe (for example when IOmniValueEnumerator is passed to the ForEach), the data will be accessed from multiple threads at the same time.

Parallel For is demoed in projects 35_ParallelFor, 36_ParallelAggregate, 37_ParallelJoin and 38_OrderedFor and its functioning is covered by

blog post www.thedelphigeek.com/2010/06/omnithreadlibrary-20-sneak-preview-1.html and by the “implementation trilogy” www.thedelphigeek.com/2011/01/parallel-for-implementation-1-overview.html (overview), www.thedelphigeek.com/2011/01/parallel-for-implementation-2-input.html(input), and www.thedelphigeek.com/2011/02/parallel-for-implementation-3-output.html (output).

Low Level Multithreading

OmniThreadLibrary started as a low-level multithreading library. It was only later that support for high-level multithreading primitives was added. Although the focus of today’s presentation is on a high-level tools I should at least mention low-level primitives that made all high-level stuff possible.

Messaging

OmniThreadLibrary tries to move as much away from the *shared data* approach as possible. Instead of that, cooperation between threads is achieved with messaging.

All data in the OmniThreadLibrary is passed around as a TOmniValue record, which is in functionality similar to Delphi’s Variant or TValue except that it’s faster. It can contain any scalar type (integer, real, TDateTime ...), strings of any type, objects and interfaces.

For more information read: www.thedelphigeek.com/2010/03/speed-comparison-variant-tvalue-and.html.

Communication between threads is implemented with TOmniMessageQueue, which passes (message ID, message data) pairs over the O(1) enqueue and dequeue, fixed-size, microlocking queue TOmniBoundedQueue. Its implementation is described in www.thedelphigeek.com/2008/07/omnithreadlibrary-internals.html.

For higher-level programming, bounded queues are not so limited and that’s why I developed TOmniQueue, a dynamically allocated, O(1) enqueue and dequeue, threadsafe, microlocking queue (yes, I’m very proud of it ;)). You can think of it as of a very fast single-ended queue that can also be used in single-thread environment. It’s internals are described in blog post www.thedelphigeek.com/2010/02/dynamic-lock-free-queue-doing-it-right.html.

Maybe the most useful queue-like tool of them all is TOmniBlockingCollection. It mimics .Net Framework 4’s

BlockingCollection (msdn.microsoft.com/en-us/library/dd267312 (VS.100).aspx). The blocking collecting is exposed as an interface that lives in the OtlCollections unit.

```
IOmniBlockingCollection = interface(IGpTraceable)
    [{208EFA15-1F8F-4885-A509-B00191145D38}]
    procedure Add(const value: TOmniValue);
    procedure CompleteAdding;
    function GetEnumerator: IOmniValueEnumerator;
    function IsCompleted: boolean;
    function Take(var value: TOmniValue): boolean;
    function TryAdd(const value: TOmniValue): boolean;
    function TryTake(var value: TOmniValue; timeout_ms: cardinal = 0): boolean;
end;
```

The blocking collection works in the following way:

- *Add* will add new value to the collection (which is internally implemented as a queue (FIFO, first in, first out)).
- *CompleteAdding* tells the collection that all data is in the queue. From now on, calling *Add* will raise an exception.
- *TryAdd* is the same as *Add* except that it doesn't raise an exception but returns False if the value can't be added.
- *IsCompleted* returns True after the *CompleteAdding* has been called.
- *Take* reads next value from the collection. If there's no data in the collection, *Take* will block until the next value is available. If, however, any other thread calls *CompleteAdding* while the *Take* is blocked, *Take* will unblock and return False.
- *TryTake* is the same as *Take* except that it has a timeout parameter specifying maximum time the call is allowed to wait for the next value.
- Enumerator calls *Take* in the *MoveNext* method and returns that value. Enumerator will therefore block when there is no data in the collection. The usual way to stop the enumerator is to call *CompleteAdding* which will unblock all pending *MoveNext* calls and stop enumeration.

A longer treatise on blocking collection (together with a very interesting example) is available at www.thedelphigeek.com/2010/02/three-steps-to-blocking-collection-3.html.

Tasks

In OTL you don't create threads but *tasks*. A task can be executed in a new thread (as I did in the demo program testOTL) or in a thread pool.

A task is created using `CreateTask`, which takes as a parameter a global procedure, a method, an instance of `TOmniWorker` class (or, usually, a descendant of that class) or an anonymous procedure (in Delphi 2009 and newer). `CreateTask` returns an interface, which can be used to control the task. As (almost) all methods of this interface return *Self*, you can chain method calls in a fluent way. The code fragment above uses this approach to declare a message handler (a method that will be called when the task sends a message to the owner) and then starts the task. In OTL, a task is always created in suspended state and you have to call `Run` to activate it.

Thread Pool

Because starting a thread takes noticeable amount of time, `OmniThreadLibrary` supports concept of *thread pools*. A thread pool keeps threads alive even when they are not used so a task can be started immediately if such thread is waiting for something to do.


Thread pool in `OmniThreadLibrary` supports automatic thread creation and destruction with user settable parameters such as maximum number of threads and maximum inactivity a thread is allowed to spend in idle state.

Using thread pool instead of “normal” thread is simple – just call `Schedule` on the task control interface instead of `Run`.

When To Use Multithreading?

The most common case is probably a slow program. You just have to find a way to speed it up. If that's the case we must somehow split the slow part into pieces that can be executed at the same time (which may be very hard to do) and then put each such piece into one thread. If we are very clever and if the problem allows that, we can even do that dynamically and create as many threads as there are processing units.

Another good reason to implement more than one thread in a program is to make it more responsive. In general, we want to move lengthy tasks away from the thread that is serving the graphical interface (GUI) into threads that are not interacting with the user (i.e. *background* threads). A good candidate for such background processing are long



database queries, lengthy imports and exports, long CPU-intensive calculations, file processing and more.

Sometimes, multithreading will actually simplify the code. For example, if you are working with an interface that has simple synchronous API (start the operation and wait for its result) and complicated asynchronous API (start the operation and you'll somehow be notified when it is completed) as are file handling APIs, sockets etc, it is often simpler to put a code that uses synchronous API into a separate thread than to use asynchronous API in the main program. If you are using some 3rd party library that only offers you a synchronous API you'll have no choice but to put it into a separate thread.

A good multithreading example is server that can serve multiple clients. Server usually takes a request from the client and then, after some potentially lengthy processing, returns a result. If the server is single-threaded, the code must be quite convoluted to support multiple simultaneous clients. It is much simpler to start multiple threads, each to serve one client.

Testing

When you're writing multithreaded applications a proper approach to testing will (and please note that I'm not using "may" or "can"!) mean a difference between a working and crashing code.

Always write automated stress tests for your multithreaded code. Write a testing app that will run some (changeable) number of threads that will execute your code for some prolonged time and then check the results, status of internal data structures, etc. – whatever your multithreaded code is depending upon. Run those tests whenever you change the code. Run them for long time – overnight is good.

Always test multithreaded code on small and large number of threads. Always test your apps with minimum number of required threads (even one, if it makes sense) on only one core and then increase number of threads and cores until your running many more threads than you have cores. I've found out that most problems occur when threads are blocked at "interesting" points in the execution and the simplest way to simulate this is to overload the system by running more threads than there are cores.

When you find a problem in the application that the automated test didn't find, make sure that you first understand how to repeat the problem. Include it in the automated test next and only then start to fix it.

In other words – unit testing is your friend. Use it!



Application design

Most bugs in multithreaded programs spring from too complicated designs. Complicated architecture equals complicated and hard to find (and even harder to fix) problems. Keep it simple!

Instead of inventing your own multithreaded solutions, use as many well-tested tools as possible. More users = more found bugs. Of course, you should make sure that your tools are regularly upgraded and that you're not using some obsolete code that everybody has run away from.

Keep the interaction points between threads simple, small and well defined. That will reduce the possibility of conflicts and will simplify the creation of automated tests.

Share as little data as possible. Global state (shared data) requires locking and is therefore bad by definition. Message queues will reduce possibility for deadlocking. Still, don't expect message-based solutions to be magically correct – they can still lead to locking.

And besides everything else – have fun! Multithreaded programming is immensely hard but is also extremely satisfying.