MULTITHREADING

FAST PROGRAMS FOR MODERN COMPUTERS
I. Basics
WHAT?

- The art of doing multiple things at the same time
WHY?

HOW?

- TThread
- System.Threading [XE7]
- OmniThreadLibrary [Windows, VCL]
WHEN?

- Slow background process
- Background communication
- Executing synchronous API
- Multicore data processing
- Multiple clients
THREADING

- A thread is a line of execution through a program
  - There is always one thread
- Multitasking (and multithreading)
  - Cooperative
    - Win 3.x
  - Preemptive
    - Time slicing
    - Parallel execution
PROCESSES VS. THREADS

- Pros
  - Processes are isolated – data protection is simple

- Cons
  - Processes are isolated – data sharing is complicated
  - Processes are heavy, threads are light
PROBLEMS

- Data sharing
  - Messaging
  - Synchronization
- Synchronization causes
  - Race conditions
  - Deadlocking
  - Livelocking
- Slowdown
FOUR PATHS TO MULTITHREADING - 1

- The Windows Way
  - FHandle := `BeginThread(nil, 0, @ThreadProc, Pointer(Self), 0, FThreadID);`
The Delphi Way

- Focus on threads
- TMyThread = class(TThread)
  procedure Execute; override;
The XE7 Way

- Focus on tasks
- task := TTask.Create(procedure begin ... end);
- future := TTask.Future<Integer>(function: Integer ...);
- TParallel.For(1, Max, procedure (I: Integer) ...);
The OmniThreadLibrary Way

- task := CreateTask(worker, ‘name’);
- Async(procedure begin ... end).Await(procedure ...);
- Parallel.For(1, 100000).Execute(procedure (i: integer) ...);
Task is part of code that has to be executed

Thread is the execution environment
THREAD POOLING

- Starting up a thread takes time
- Thread pool keeps threads alive and waits for tasks
- Automatic thread startup/shutdown
FThread1 := TTestThread1.Create;

FThread1.Terminate;
FThread1.WaitFor;
FreeAndNil(FThread1);

FThread2 := TTestThread2.Create(true);
FThread2.FreeOnTerminate := true;
FThread2.OnTerminate := ReportThreadTerminated;
procedure TTestThread1.Execute;
begin
  while not Terminated do begin
    // some real work could be done here
    end;
end;
TTHREAD EXTRAS

- CreateAnonymousThread
- Synchronize, Queue
- ReturnValue
- FatalException
- Handle, ThreadID, Priority
Pros
- Low-level approach offers full execution speed
- Multi-OS support

Cons
- Offers no help to simplify multithreading programming
DELPHI XE7 DEMO
Encapsulates a task (a work to be done)
Runs in a thread pool
TTask.Create + ITask.Start
TTask.Run
ITask.Wait/TTask.WaitForAll/TTask.WaitForAny
No OnTerminate notification
FUTURE

- Performs a computation in background and returns a result
- TTask.Future<ReturnType>
- IFuture<ReturnType>.Value
- IFuture<ReturnType>.Status
PARALLEL FOR

- TParallel.For(lowBound, highBound, workerProc);
- Watch for shared memory access!
PROS AND CONS

- **Pros**
  - Simple usage
  - Hard parts are already implemented
  - Multi-OS support

- **Cons**
  - Limited functionality
  - No messaging
II. Do’s and Don’t’s
SHARED MEMORY

- Read / Modify / Write
  - Increment / Decrement
  - Simultaneously reading and writing into a list
    - TList, TStringList, TList<T>, ...
  - Arrays are usually fine
    - Don’t access same element from two threads
    - Element size ≥ SizeOf(pointer)
ATOMIC CHANGES

- SyncObjs
- Locking
  - TCriticalSection
  - TSpinLock
  - TMultiReadExclusiveWriteSynchronizer / TMREWSync (SysUtils)
- “Interlocked” operations
  - TInterlocked
PROBLEMS CAUSED BY LOCKING

- Deadlocks
- Livelocks
- Slowdown
SyncObjs
TThreadList
TThreadedQueue
TMonitor
   Be careful!

threadvar
MECHANISMS

- TEvent
- Messages [Windows]
- TCP
- Shared memory (with atomic changes)
  - Message queue
III. OmniTHREADLIBRARY
OMNITHREADLIBRARY IS ...

- ... VCL for multithreading
  - Simplifies programming tasks
  - Componentizes solutions
  - Allows access to the bare metal
- ... trying to make multithreading possible for mere mortals
- ... providing *well-tested* components packed in *reusable* classes with *high-level* parallel programming support
PROJECT STATUS

- http://www.omnithreadlibrary.com
- Delphi 2007 →
- OpenBSD license
- Actively used
- https://leanpub.com/omnithreadlibrary
- http://www.omnithreadlibrary.com/webinars.htm
- Google+ community
Checkout / Download + Unpack
Add *path* & *path/src* to search path
*uses Otl*
ABSTRACTION LAYERS

- Low-level
  - TThread replacement
  - Similar to TTask [XE7]
  - Communication

- High-level
  - Requires Delphi 2009
  - “Multithreading for mere mortals”
  - ‘Parallel for’ and much more
LOW-LEVEL MULTITHREADING
CREATING A TASK

- `CreateTask(task_procedure)`
- `CreateTask(task_method)`
- `CreateTask(TOmniWorker_object)`
- `CreateTask(anonymous_procedure)`
MESSAGING

- Messaging preferred to locking

- TOmniMessageQueue

- TOmniQueue
  - Dynamically allocated, O(1) enqueue and dequeue, threadsafe, microlocking queue

- TOmniBlockingCollection

- TOmniValue
FHelloTask := CreateTask(TAsyncHello.Create(), 'Hello')
    .SetParameter('Delay', 1000)
    .SetParameter('Message', 'Hello')
    .OnMessage(Self)
    .OnTerminated(
        procedure
            begin
                lbLog.Items.Add('Terminated');
            end)
    .Run;
LOW-LEVEL CLASSES

- OtlTask
  - IOmniTask
- OtlTaskControl
  - IOmniTaskControl
- OtlCommon
  - TOmniValue
  - Environment
- OtlContainers
  - TOmniBoundedStack
  - TOmniBoundedQueue
  - TOmniQueue
- OtlSync
  - TOmniCS
  - TOmniMREW
  - Locked<T>
IV. High-Level Mulithreading
ABSTRACTIONS

- Async/Await
- Async
- Future
- ForEach / For
- Join
- Parallel task
- Background worker
- Pipeline
- Fork/Join
- `Parallel.Async(code)`
ASYNC/AWAIT

- Simplified syntax
- Async(TProc).Await(TProc);
FUTURE

- Wikipedia
  - “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.”
- Start background calculation, wait on result.
Future := Parallel.Future<
\textit{type}> (\textit{calculation});
\[\text{Value} := \text{Future.Value};\]
FOREACH / FOR

- `Parallel.ForEach(from, to).Execute(
  procedure (const value: integer);
  begin
    //...
    end);

- `Parallel.ForEach(source).Execute(
  procedure (const value: TOmniValue) ...`

- `Parallel.ForEach<string>(source).Execute(
  procedure (const value: string) ...`
ForEach
(source, code)

output

code

code

code

source

Optional
Parallel.Join([task1, task2, task3, ... taskN]).Execute
PARALLEL TASK

- `Parallel.ParallelTask.Execute(code)`
Client/Server
Parallel.Pipeline([stage1, stage2, stage3]).Run
while <has data>
    read <data>
    compress <data>
    encrypt <data>
    write <data>

while <has data>
    read <data> into <read-buf>
    compress <read-buf> into <compress-buf>
    encrypt <compress-buf> into <encrypt-buf>
    write <encrypt-buf>

while <has data>
    read <data> into <read-buf>
    compress <read-buf> into <compress-buf>
    encrypt <compress-buf> into <encrypt-buf>
    write <encrypt-buf>

while <has data>
    read <data>
    insert <data> into <compress queue>
    read <data-c> from <compress queue>
    ...    encrypt <data-e>
    insert <data-e> into <write queue>
    read <data-w> from <write queue>
    write <data-w>

while <has data>
    read <data>
    insert <data> into <compress queue>
while read <compress queue>
    compress ... <encrypt queue>
    encrypt <data>
    insert <data> into <write queue>
while read <write queue>
    write <data>

while <has data>
    read <data>
    insert <data> into <compress queue>
while read <compress queue>
    compress <data>
    insert <data> into <encrypt queue>
while read <encrypt queue>
    encrypt <data>
    insert <data> into <write queue>
while read <write queue>
    write <data>
var
    pipeOut: IOmniBlockingCollection;

pipeOut := Parallel.Pipeline
    .Stage(StageGenerate)
    .Stage(StageMult2)
    .Stage(StageSum)
    .Run;
FORK/JOIN

- Divide and Conquer
FORK/JOIN

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

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

Result := Max(max1.Value, max2.Value);
WORDS OF (HARD LEARNED) WISDOM
“New programmers are drawn to multithreading like moths to flame, with similar results.”

- Danny Thorpe
KEEP IN MIND

- Never use VCL from a background thread!
- Don’t parallelize everything
- Don’t create thousands of threads
- Rethink the algorithm
- Prove the improvements
- Test, test and test
BE AFRAID

- Designing parallel solutions is hard
- Writing multithreaded code is hard
- Testing multicore applications is hard
- Debugging multithreading code is pure insanity
  - Debugging high-level abstractions is just hard
QUESTIONS?