Parallel programming in Pascal

Primož Gabrijelčič

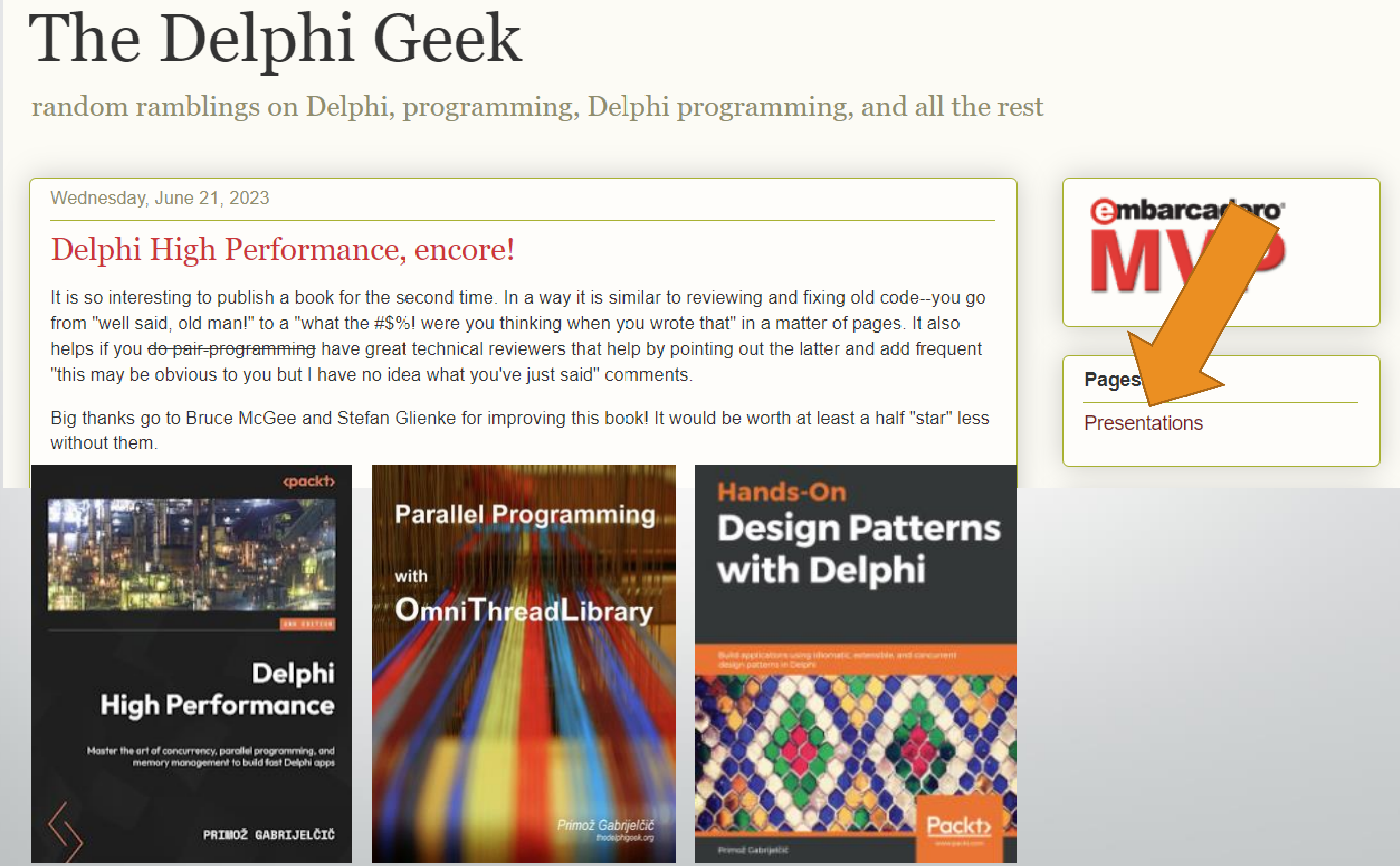
About me

- programmer, MVP, writer, blogger, consultant, speaker

Blog	*http://thedelphigeek.com*
Twitter	*@thedelphigeek*
Skype	*gabr42*
LinkedIn	*gabr42*
GitHub	*gabr42*
SO	*gabr*

Professional path

- 198x high school
  HP 41C, ZX Spectrum [*HiSoft Pascal*], PDP-11
- 199x university, Monitor magazine
  CP/M [*Turbo Pascal 3+*], VAX/VMS [*VAX Pascal*, *Perl*],
  DOS [*Turbo/Borland Pascal 4+*], OS/2, Windows [*Delphi 2*]
- 20xx The Delphi Magazine, Blaise Pascal Magazine, The Delphi Geek,
  books (Packt Publishing, self-published)
  R&D Manager @ FAB: high-performance parallel systems
  Windows [*Delphi*, *JavaScript*, *Python*]



The Delphi Geek

random ramblings on Delphi, programming, Delphi programming, and all the rest

Wednesday, June 21, 2023

Delphi High Performance, encore!

It is so interesting to publish a book for the second time. In a way it is similar to reviewing and fixing old code--you go from "well said, old man!" to a "what the #$%! were you thinking when you wrote that" in a matter of pages. It also helps if you ~~do pair-programming~~ have great technical reviewers that help by pointing out the latter and add frequent "this may be obvious to you but I have no idea what you've just said" comments.

Big thanks go to Bruce McGee and Stefan Glienke for improving this book! It would be worth at least a half "star" less without them.

Pages

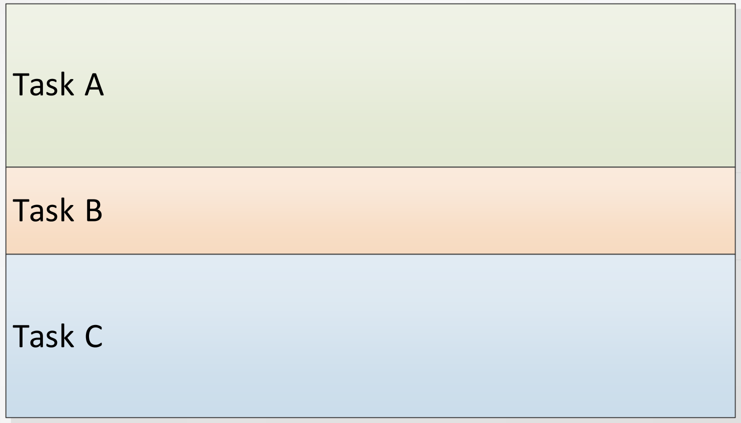
Presentations

Multithreading

Prologue

From one to many

- Single-tasking

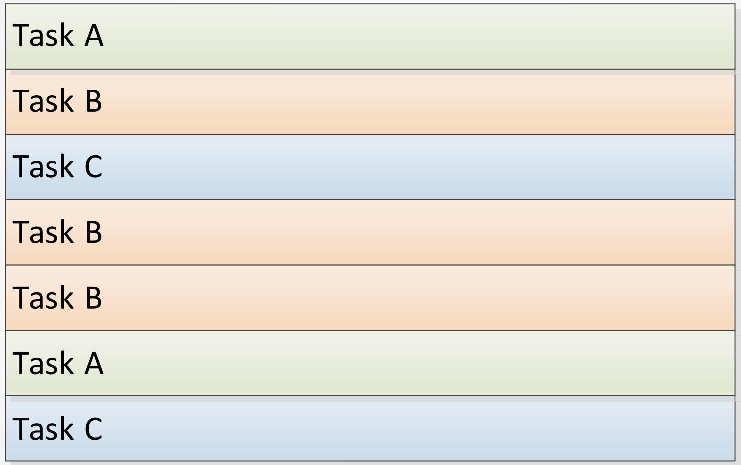

Task A

Task B

Task C

From one to many

- Single-tasking
- Multi-tasking
  - Cooperative
  - Preemptive



Task A

Task B

Task C

Task B

Task B

Task A

Task C

From one to many

- Single-tasking
- Multi-tasking
  - Cooperative
  - Preemptive
- Multi-threading
  - Single CPU

Task A / thread 1

Task B / thread 1

Task C / thread 2

Task B / thread 2

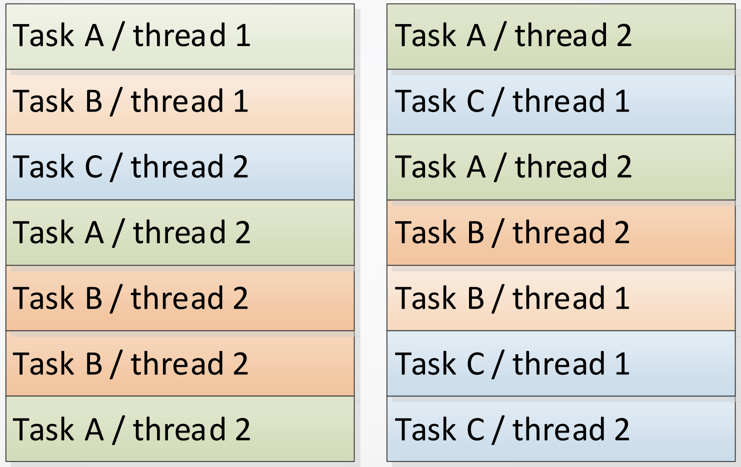
Task B / thread 2

Task A / thread 2

Task C / thread 2

From one to many

- Single-tasking
- Multi-tasking
  - Cooperative
  - Preemptive
- Multi-threading
  - Single CPU
  - Multiple CPUs



Task A / thread 1	Task A / thread 2
Task B / thread 1	Task C / thread 1
Task C / thread 2	Task A / thread 2
Task A / thread 2	Task B / thread 2
Task B / thread 2	Task B / thread 1
Task B / thread 2	Task C / thread 1
Task A / thread 2	Task C / thread 2

Processes vs. threads

Process

- A collection of program's resources
  - Allocated memory
  - File handles
  - Sockets
  - UI elements
- Memory & resource protection
- "Heavy"

Thread

- Execution state
  - Execution address
  - CPU registers
  - Stack
- Memory & resource sharing
- "Light"

Why?

- Responsiveness (non-blocking UI)
- Faster program execution
  - Handling multiple clients
  - Faster data processing

How?

- OS
  - CreateThread, pthread_create …
- Compiler
  - async … await [.NET …]
- RTL
  - BeginThread, TThread

Problems!

- Sharing data
  - Simultaneous writing

```
FData: integer;
```

```
FData := FData + 1;


tmp := FData;
tmp := tmp + 1;
FData := tmp;
```

```
FData := FData + 1;


tmp := FData;
tmp := tmp + 1;
FData := tmp;
```

Problems!

- Sharing data
  - Simultaneous writing
  - Simultaneous reading and writing

```
               FData: TList<T>;

for var t in FData do     FData.Delete(0);
  Process(t);


                          FData.Add(t);
```

Problems!

- Sharing data
  - Simultaneous writing
  - Simultaneous reading and writing
  - Creating/destroying shared objects/interfaces

```
FLazy: TLazy;
```

```
FLazy := TLazy.Create;


if not assigned(FLazy)
then
  FLazy := TLazy.Create;
```

```
FLazy := TLazy.Create;


if not assigned(FLazy)
then
  FLazy := TLazy.Create;
```

Problems!

- Sharing data
  - Simultaneous writing
  - Simultaneous reading and writing
  - Creating/destroying shared objects/interfaces
  - Hidden behaviour

```
function TStream.GetSize: Int64;
var
  Pos: Int64;
begin
  Pos := Seek(0, soCurrent);
  Result := Seek(0, soEnd);
  Seek(Pos, soBeginning);
end;

function TCustomMemoryStream.Seek(const Offset: Int64;
  Origin: TSeekOrigin): Int64;
begin
  case Origin of
    soBeginning: FPosition := Offset;
    soCurrent: Inc(FPosition, Offset);
    soEnd: FPosition := FSize + Offset;
  end;
  Result := FPosition;
end;
```

Solutions

- Synchronization (locking)

```
Fcs: TCriticalSection;
Fdata: integer;
```

```
Fcs.Acquire;
try
  FData := FData + 1;
finally
  Fcs.Release;
end;
```

```
Fcs.Acquire;
try
  FData := FData + 1;
finally
  Fcs.Release;
end;
```

Solutions

- Synchronization (locking)
  - Not enforced!

```
Fcs: TCriticalSection;
Fdata: integer;
```

```
Fcs.Acquire;
try
  FData := FData + 1;
finally
  Fcs.Release;
end;
```

```
FData := FData + 1;
```

Solutions

- Synchronization (locking)
  - Not enforced!
  - Deadlocks

```
  Fcs1: TCriticalSection;
  Fcs2: TCriticalSection;

Fcs1.Acquire;  |  Fcs2.Acquire;
Fcs2.Acquire;  |  Fcs1.Acquire;
```

Solutions

- Synchronization (locking)
  - Not enforced!
  - Deadlocks
  - Slower execution
    - Keep locked areas as short as possible!

Solutions

- Synchronization (locking)
- Interlocked operations
  - AtomicIncrement [System]
  - InterlockedIncrement [Windows]
  - TInterlocked.Increment [SyncObjs]

```
                 FData: integer;

AtomicIncrement(FData);  |  AtomicIncrement(FData);
```

Solutions

- Synchronization (locking)
- Interlocked operations
  - Not enforced!

```
                 FData: integer;

AtomicIncrement(FData);    Inc(FData);
```

Solutions

- Synchronization (locking)
- Interlocked operations
  - Not enforced!
  - Faster
  - Limited
  - Hard to use

Testing

- Extremely hard to test
- “Infinite” possible interactions between threads
- Stress-testing

```
          FData: integer;

tmp := FData;      tmp := FData;
tmp := tmp + 1;    tmp := tmp + 1;
FData := tmp;      FData := tmp;
```

Alternatives

- Multiprocessing
  - OpenMP
- GPU
  - OpenCL
  - C/C++
- Clusters, grids, networks

Threads

Act 1 - Past

Threads

- Delphi 2
- BeginThread
- TThread
    - Start thread / Main thread loop / Terminate thread
- Synchronization
    - OS: Critical section, Mutex, Semaphore, Event

Problems

- Multithreaded code written “from scratch”
- 1000 different ways and 1000^2 different bugs
- No support for communication
- Very limited support for synchronization

Tasks

Act 2 - Present

Tasks

- .NET 4 Task Parallel Library
  - Tasks, Concurrent Collections, Cancellation, Parallel For, LINQ
  - C# async/await
- Thread = operating system concept
  - You tell the system **how** to do the work
  - Usually: A new thread each time
- Task = part of code
  - You tell the library **what** you want to execute in parallel
  - Usually: threads come from a thread pool
    - Reason: thread creation takes time

Synchronization mechanisms

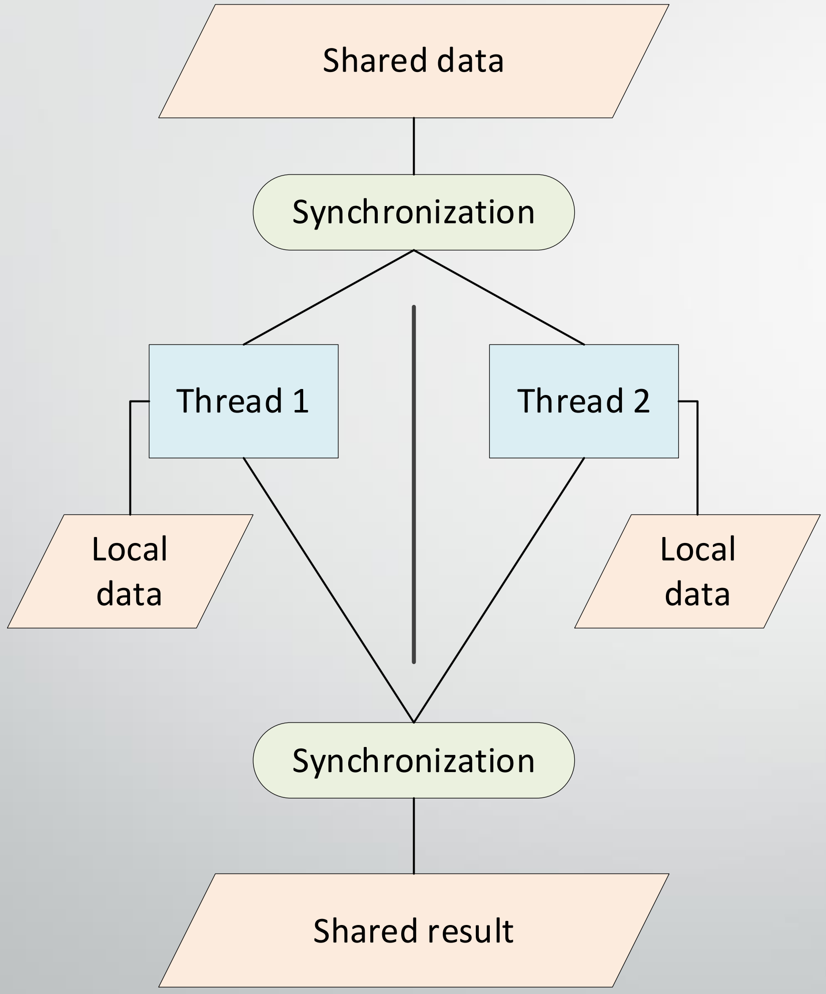
- RTL
    - TMonitor
        - Spin-lock each object
    - TThread.Synchronize
- OS
    - Readers/writer [SRW, pthread_rwlock]
    - Condition variables [TRTLConditionVariable, pthread_cond_t]

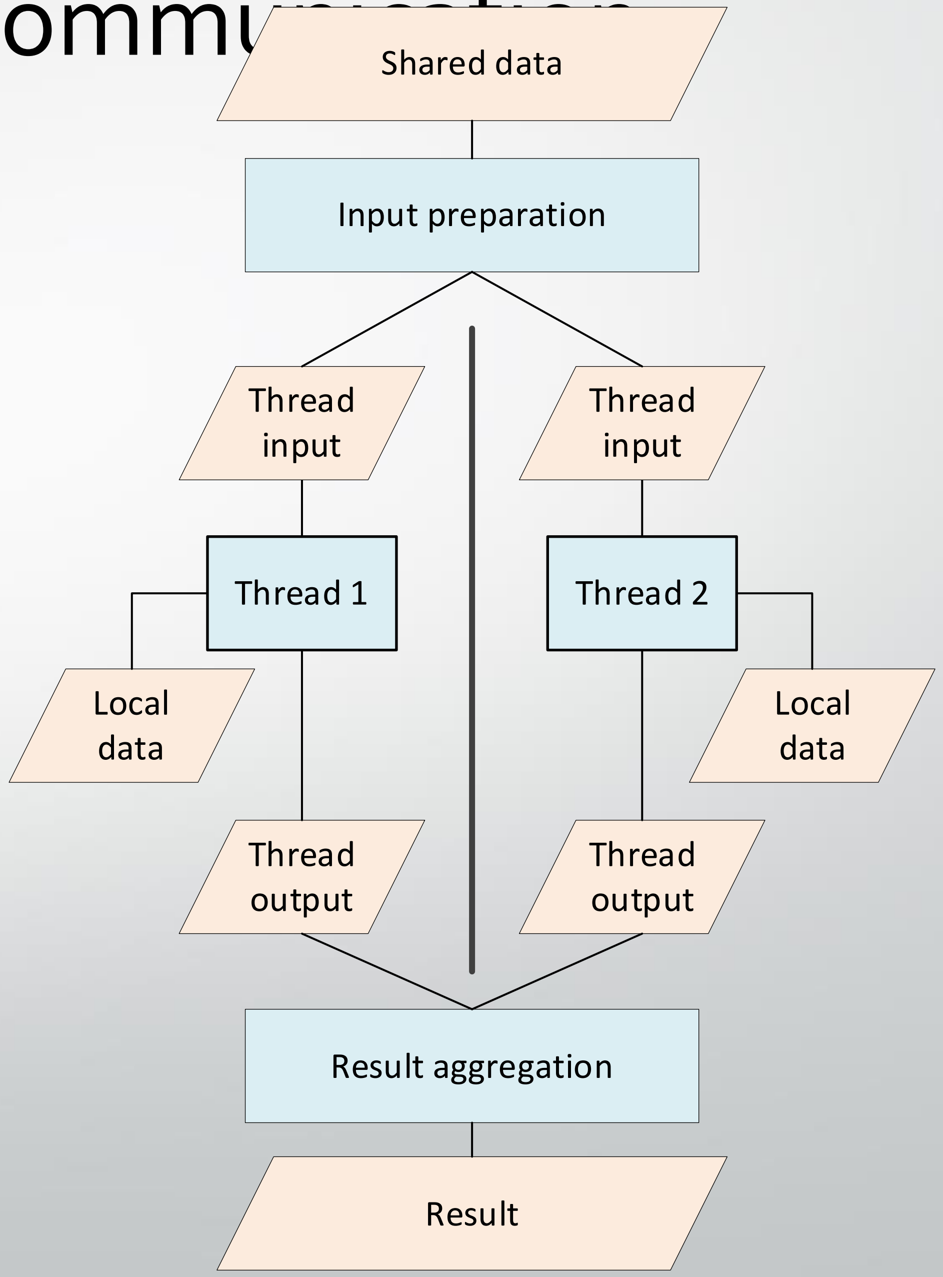
Communication mechanisms

- OS messages [Windows]
- TThread.Queue, ForceQueue
- Polling
- IOmniBlockingCollection
- Locking is acceptable here
  - Too slow? Reduce number of messages!

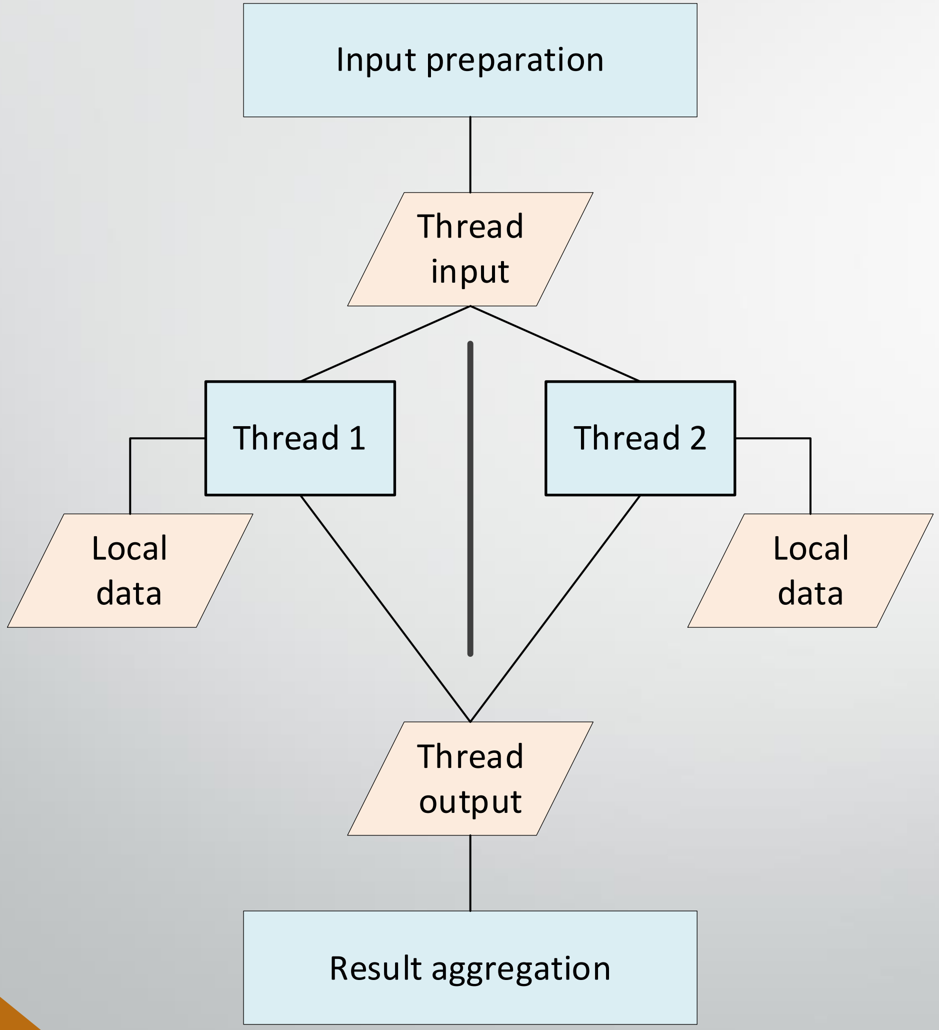
- Locking + shared list
- Interlocked + shared lists

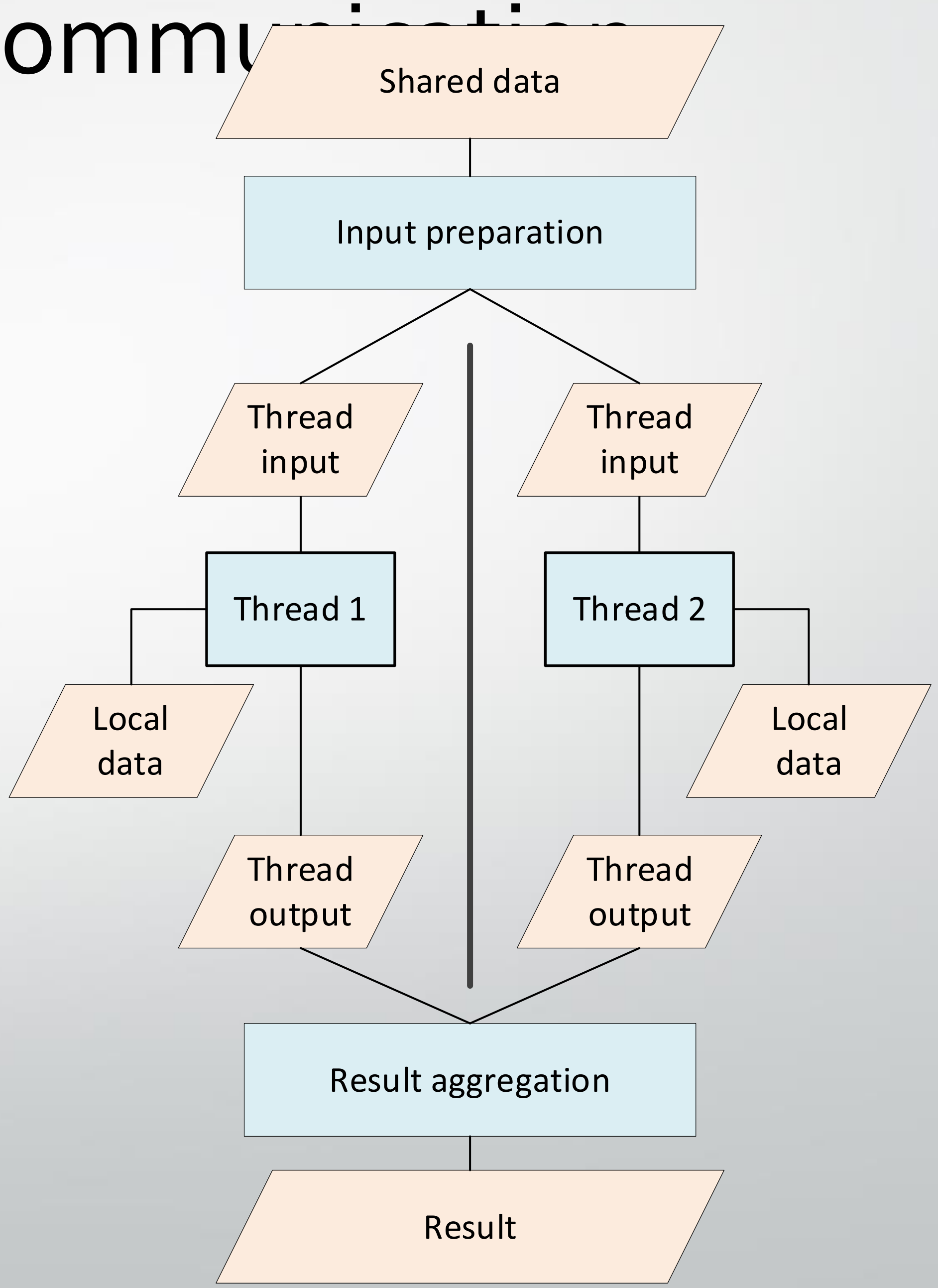
Synchronization vs. communication





Synchronization vs. communication





Patterns

Act 3 - Future

Patterns

- Pre-packaged solutions to frequent problems
- All thread/task management is hidden behind a *façade* pattern

- Stop caring about task management, focus on the problem
- Pick a right pattern and write single-threaded code; library will do the rest
- Your code is testable; library is as simple as possible and well-tested

Patterns

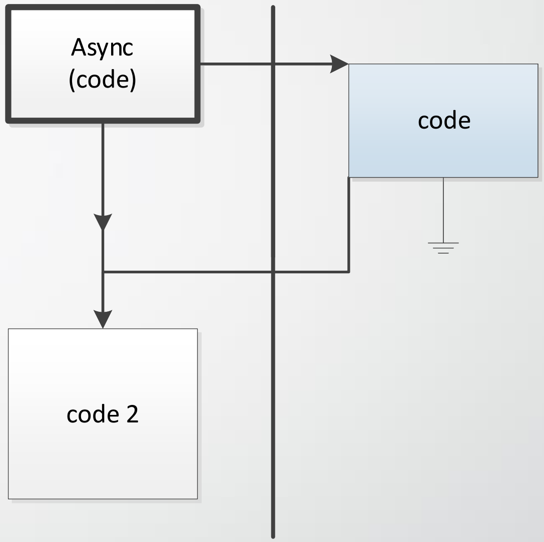
- Parallel Programming Library
- OmniThreadLibrary

Async/Await

- *Execute code in a worker thread*
  - Optionally execute more code in the main thread after that is done

-
  ```
  Async(
    procedure begin
      DoBackgroundWork;
    end)
  ```

  ```
  .Await(
    procedure begin
      UpdateUI;
    end)
  ```

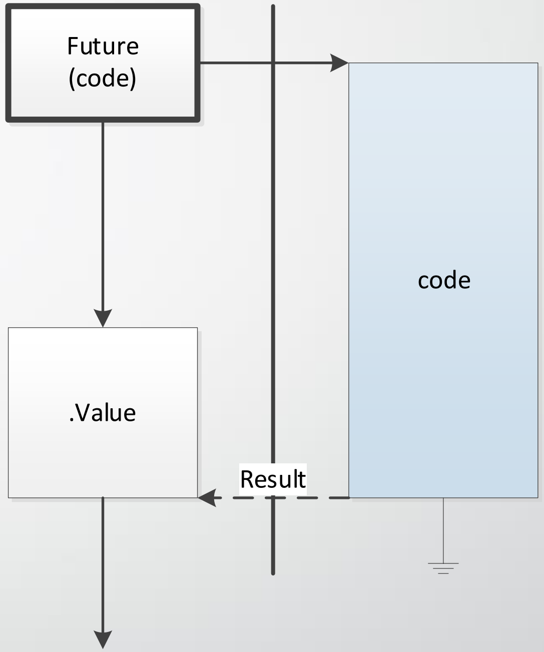


Async

- TThread.CreateAnonymousThread
- multithreadprocs / threadPool.DoParallel

Future

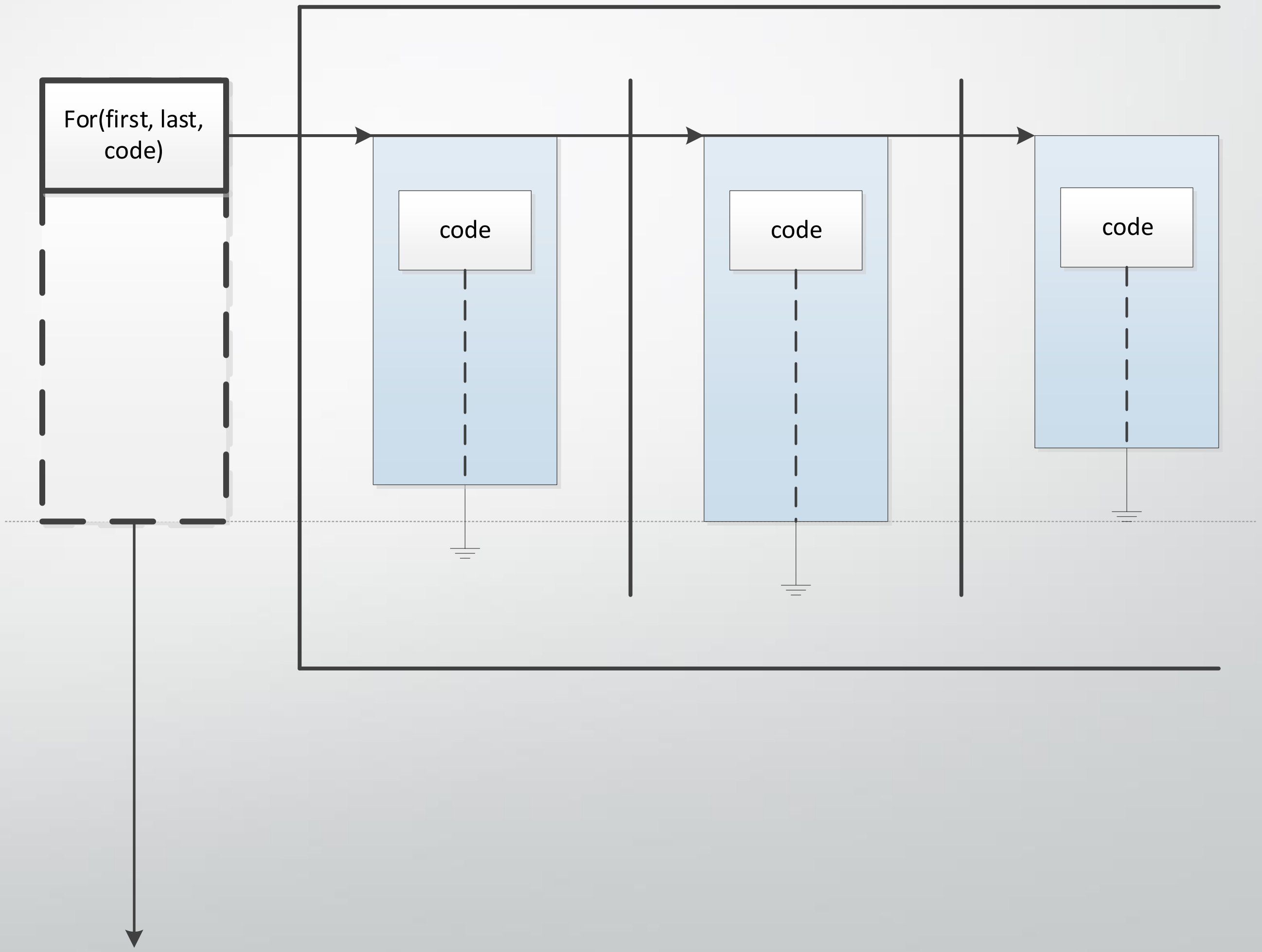
- *Start background calculation, later retrieve the result*
- `FCalculation := TTask.Future<integer>(Calculate);`
- `ShowResult(FCalculation.Value);`
- `FCalculation := nil;`



Parallel For

- *Iterate over a range in parallel*

-
  ```
  TParallel.For(1, 1000,
    procedure(i: integer)
    begin
      ProcessIndex(i);
    end);
  ```
- Simple but dangerous!



Parallel For

```
for i := 2 to CHighestNumber do
  if IsPrime(i) then
    Inc(count);
```

```
TParallel.For(2, CHighestNumber,
  procedure (i: integer)
  begin
   if IsPrime(i) then
     Inc(count);
  end);
```

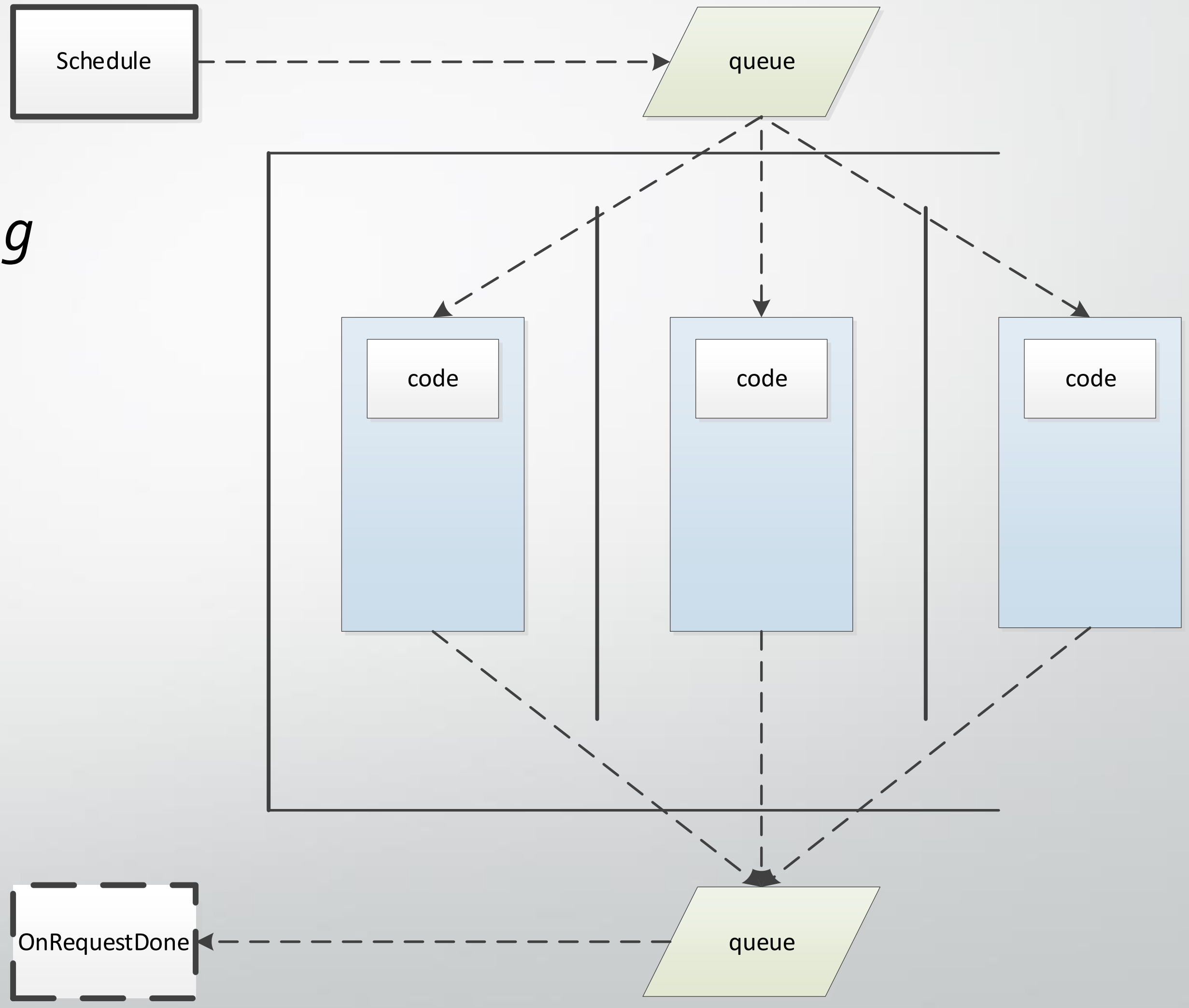
Parallel For

```
for i := 2 to CHighestNumber do
  if IsPrime(i) then
    Inc(count);
```

```
TParallel.For(2, CHighestNumber,
  procedure (i: integer)
  begin
   if IsPrime(i) then
     ~~Inc(count);~~
     TInterlocked.Increment(count);
  end);
```

Background Worker

- *Start a background data processing server, optionally running on multiple threads*

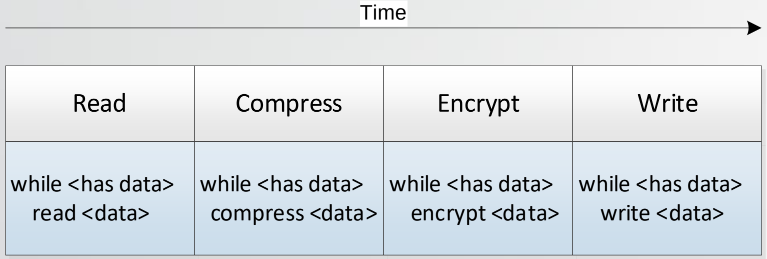


Background Worker

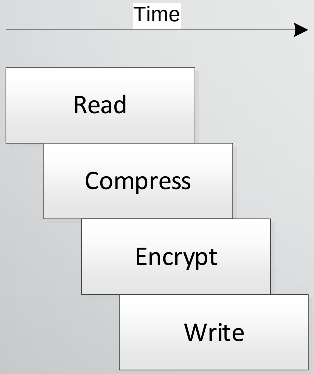
```
FBackgroundWorker := Parallel.BackgroundWorker.NumTasks(2)
 .Execute(
   procedure (const workItem: IOmniWorkItem)
   begin
     workItem.Result := ProcessData(workItem.Data);
   end )
 .OnRequestDone(
   procedure (const Sender: IOmniBackgroundWorker;
     const workItem: IOmniWorkItem)
   begin
     DisplayResult(workItem.Result);
   end;
```

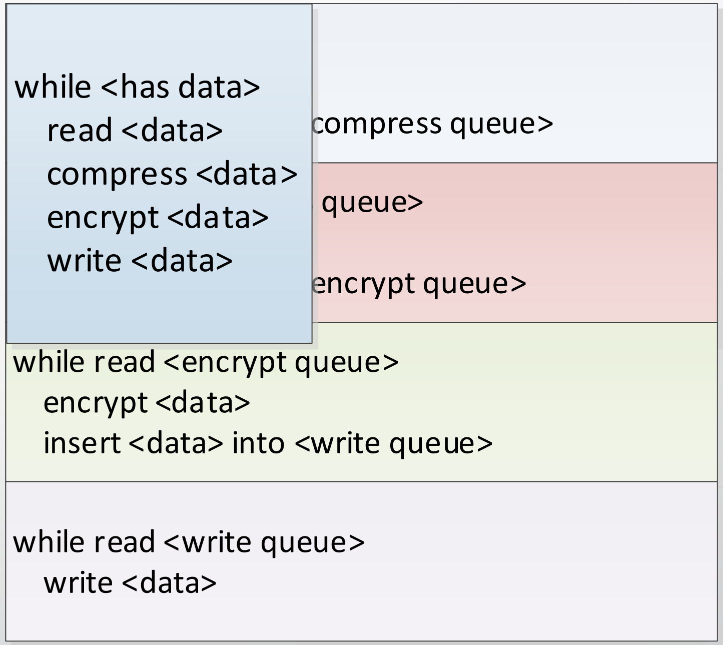
Pipeline

- *Process data in multiple (overlapping) stages*



Read	Compress	Encrypt	Write
while <has data> read <data>	while <has data> compress <data>	while <has data> encrypt <data>	while <has data> write <data>





```
while <has data>
    read <data>
    compress <data>
    encrypt <data>
    write <data>
```

```
compress queue>
```

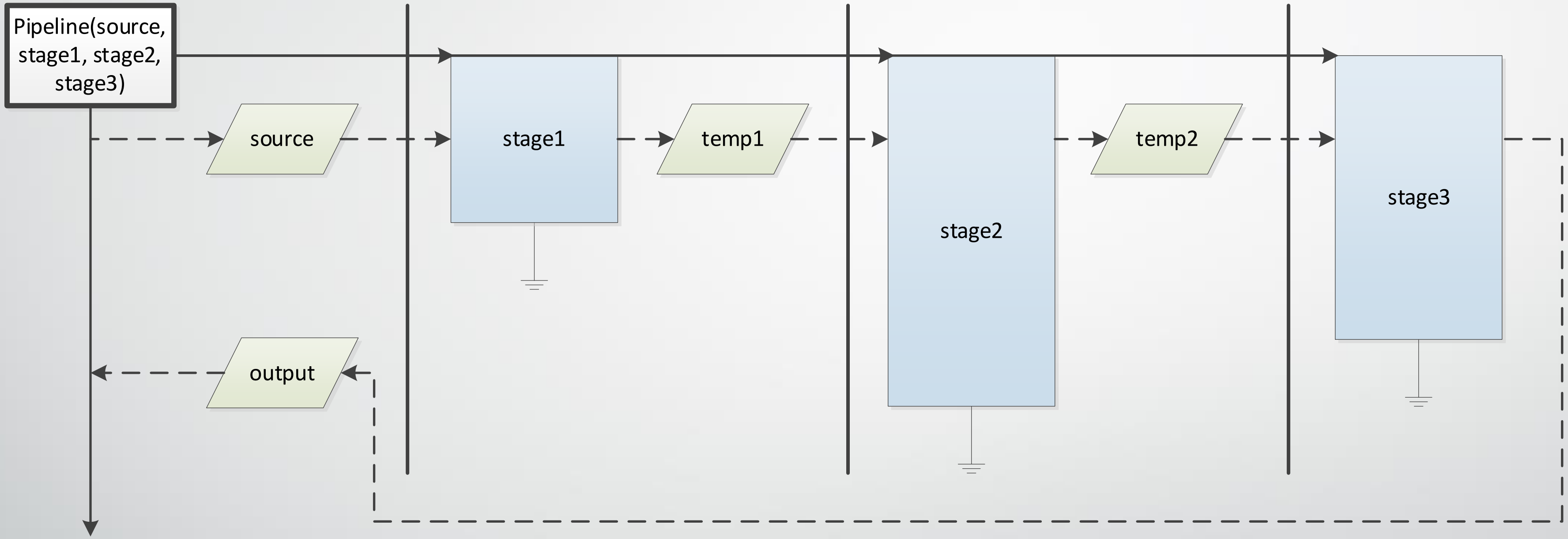
```
queue>

encrypt queue>
```

```
while read <encrypt queue>
   encrypt <data>
   insert <data> into <write queue>
```

```
while read <write queue>
   write <data>
```

Pipeline

- *Process data in multiple (overlapping) stages*



- `Pipeline.Stage(Reader).Stage(Compressor).Stage(Encryptor).Stage(Writer).Run`

Just one more thing

Prologue

Remember!

- Access shared data in tight, well-tested code
- Use well-tested libraries, data duplication and communication!
- When in doubt, write single-threaded code!

“New programmers
are drawn to multithreading
like moths to flame,
with similar results.”

-Danny Thorpe
Chief Scientist for Windows and .NET developer tools at Borland