Unveiling ComputeCore: Your Multithreading Maestro for Responsive Delphi Applications

In the world of Delphi development, crafting applications that are both powerful and fluid can often feel like a delicate balancing act. Long-running operations, complex calculations, or network requests can easily bring a user interface to a grinding halt, leading to frustrated users and a perception of sluggishness. This is where the art of multithreading comes into play, allowing applications to perform heavy lifting in the background while keeping the user interface responsive and interactive.

Enter `ComputeCore`, a custom Delphi unit designed to simplify and manage complex multithreaded operations, enhancing application responsiveness and overall performance. It aims to provide a structured approach to overcome the inherent challenges of multithreading in Delphi, particularly given the VCL's single-threaded nature.[1] Think of

`ComputeCore` as a specialized tool in your Delphi toolkit, much like the layout systems discussed on thedelphigeek.com, designed to make a specific, often tricky, part of development easier and more robust.[3]

With the full source code of `ComputeCore.pas` now in hand, we can dive deep into its actual design and implementation, revealing the clever solutions it employs to tackle the complexities of concurrent programming.

The Vision Behind ComputeCore: Goals and Design Philosophy

Any custom unit built to tackle concurrency in Delphi must have clear objectives. For `ComputeCore`, the primary goals revolve around enhancing application quality and developer productivity:

- **Run Computationally Intensive Tasks:** The core purpose is to offload heavy computations from the main application thread, ensuring the user interface remains responsive.[1]
- **Tasks Can Spawn Other Tasks:** This is crucial for complex algorithms where a large problem is broken down into smaller, dependent sub-problems. `ComputeCore` is designed to handle this nesting gracefully.
- **Simple Interface:** The unit aims to provide an easy-to-use API, abstracting away the complexities of raw thread management.

The design philosophy of `ComputeCore` leans heavily into a **task-oriented** approach. Rather than forcing developers to directly manage `TThread` instances, it provides an interface (`ITask`) for defining "units of work," similar to the `ITask` interface found in Delphi's Parallel Programming Library (PPL).[4] This abstraction allows developers to focus on

what needs to be done, not *how* it's done concurrently.

The very existence of a custom unit like `ComputeCore`, especially when Delphi's PPL (`TTask`, `ITask`) is available [4], suggests a deliberate architectural choice. While PPL is a powerful general-purpose library, ComputeCore might offer a simpler, more opinionated API for particular use cases, or it integrates seamlessly with a legacy system that doesn't easily adopt PPL's anonymous methods. This implies a focus on addressing specific pain points or offering a tailored abstraction that the PPL, while powerful, might not perfectly fit for its original author.

Key Implementation Choices: Simplicity and Focus

ComputeCore makes two explicit design choices that simplify its internal workings and define its scope:

- **No Return Value:** The `ITask` interface and its implementation `TCCTask` are designed to execute a `TProc` (a procedure with no parameters and no return value). This means tasks don't directly return results through the `ITask` interface itself. If a task needs to communicate a result, it must do so through other means, such as updating shared data (with proper synchronization) or queuing a message back to the main thread. The `ExceptObj` property on `ITask` does, however, provide a way to retrieve any exception raised during the task's execution.
- **No Cancellation:** Unlike Delphi's PPL `ITask` which includes a `Cancel` method and `TTaskStatus.Canceled` [4],

  `ComputeCore`'s `ITask` interface does not expose a cancellation mechanism. Tasks submitted to `ComputeCore` are expected to run to completion. This simplifies the internal state management and avoids the complexities of handling graceful task termination.

ComputeCore: A Top-Level View

From a high-level perspective, `ComputeCore` operates around a central manager object, `TComputeCore`, which implements the `IComputeCore` interface.

- **One Singleton** `GlobalComputeCore: IComputeCore`**:** The unit defines a global variable `GlobalComputeCore` of type `IComputeCore`. This is the primary entry point for most applications using `ComputeCore`.
- **Automatic Creation on First Use:** By default, `GlobalComputeCore` is automatically created the first time `TTask.Run` is called. It initializes with `CPUCount - 1` worker threads, leaving one CPU core free for the main application thread or other system processes. This is an example of "optimistic initialization," where the resource is created only when needed, but with care taken to ensure thread-safe initialization using `TInterlocked.CompareExchange` to prevent multiple threads from trying to create it simultaneously.
- **Overriding Default Creation:** If you need a different number of threads or want to control the creation explicitly, you can create the `TComputeCore` instance yourself before any `TTask.Run` calls: `GlobalComputeCore := TComputeCore.Create(numThreads)`.
- `IComputeCore` **is Thread-Safe:** The `IComputeCore` interface, implemented by `TComputeCore`, is designed to be thread-safe. Its methods can be called concurrently from any thread, including other worker threads within the `ComputeCore` pool itself. This is achieved through internal synchronization mechanisms.
- **Simple Public Interface:** The `IComputeCore` interface exposes a concise public API:
  - `function Run(const taskProc: TProc): ITask; overload;`: Schedules a procedure to be executed as a task.

  - procedure WaitFor(const task: ITask);: Blocks the calling thread until the specified task completes.
- **Compatibility with PPL (**ITask **and** TTask**):** ComputeCore provides its own ITask interface and a TTask class. These are designed to be compatible with the naming conventions of Delphi's Parallel Programming Library (PPL).[4]
- TTask **Delegates to** GlobalComputeCore**:** The TTask class (which is *not* the PPL's TTask) acts as a simple wrapper. Its Run and WaitFor class methods mostly just call into the GlobalComputeCore instance. TTask.Run is also responsible for the optimistic initialization of GlobalComputeCore if it hasn't been created yet.
- **Multiple Compute Cores:** While GlobalComputeCore is the default and most convenient way to use the unit, you *could* create multiple TComputeCore instances. However, if you do, you would not be able to use the TTask.Run and TTask.WaitFor class methods, as they are hardwired to use the GlobalComputeCore singleton. You would interact directly with your custom IComputeCore instances.

Deep Dive: Data Structures within IComputeCore

The TComputeCore class, which implements IComputeCore, manages its internal state using several key data structures:

- FThreads: TArray<TCCThread>**:** This is an array that owns all the TCCThread worker objects created by the ComputeCore instance. It's preallocated to the numThreads specified during creation. This array is primarily accessed during the constructor and destructor, so it doesn't require constant protected access.
- FInactiveThreads: TArray<TCCThread>**:** This array acts as a stack to store references to worker threads that are currently idle and waiting for tasks. FInactiveTop is an integer index that points to the top of this stack. When a thread becomes inactive, it's pushed onto this stack; when a new task needs a thread, one is popped from here. This structure requires protected access due to concurrent modifications.
- FTasks: IQueue<ITask>**:** This is the central queue where all submitted tasks (ITask objects) are held, awaiting execution by a worker thread. ComputeCore uses Spring.Collections.TCollections.CreateQueue<ITask>, indicating a reliance on the Spring4D framework for its queue implementation. This queue is a critical shared resource and requires protected access.

Access to Shared Data: Navigating the Concurrency Minefield

In any multithreaded environment, **preserving data integrity** is the single biggest concern.[2] When multiple threads access and potentially modify the same shared data, chaos—in the form of race conditions and corrupted data—can quickly ensue.

ComputeCore is meticulously designed to prevent this within its own operations.

- **Protected with a** Monitor**:** TComputeCore uses a Monitor (specifically, MonitorEnter(Self) and MonitorExit(Self)) to protect access to its shared internal data. This ensures that at any given time, at most one thread can execute methods that modify the shared state.[6] The

`__Acquire` and `__Release` inline procedures encapsulate these `MonitorEnter` and `MonitorExit` calls, making the locking explicit and concise.

- **Minimize Shared Access:** A key principle for performance in multithreaded programming is to keep the work done inside a critical section (or monitor-protected block) to an absolute minimum.[2]

  `ComputeCore` adheres to this by performing only necessary operations while the lock is held.

- **Visual Separation of Methods:** The source code employs a clear convention for method naming to indicate their thread-safety characteristics:
  - **Safe Methods:** Methods that acquire exclusive access internally (using `__Acquire` and `__Release`) are public or protected and handle their own synchronization. Examples include `Run`, `WaitFor`, `ActivateInactiveThread`, `QueueTask`, and `AllocateTask`.
  - **Unsafe Methods (`_U` suffix):** Methods ending in `_U` (e.g., `GetTask_U`, `MarkThreadActive_U`) are `strict protected` and *expect* exclusive access to be already established by the caller. This visual separation is a great way to enforce correct usage and prevent accidental unsynchronized calls.
- **Specific Protected Access Needs:** Protected access is explicitly needed when interacting with the `FTasks` queue (e.g., `GetTask_U`, `QueueTask`) and the `FInactiveThreads` array (e.g., `MarkThreadActive_U`, `ActivateInactiveThread`).
- `FThreads` **Access:** The `FThreads` array, which simply holds references to the worker thread objects, is only accessed during the `TComputeCore` constructor and destructor. Since these operations happen sequentially (during initialization and shutdown), `FThreads` does not require protected access during the lifetime of the `ComputeCore` instance.

Running a Task: From Submission to Completion

The process of running a task within `ComputeCore` is designed for simplicity and efficiency:

1. **Task Creation:** When `IComputeCore.Run(const taskProc: TProc)` is called, a new `TCCTask` object is created, encapsulating the provided `taskProc` (an anonymous method). This `TCCTask` object also creates a `TEvent` (`FWorkDone`) which will be signaled when the task completes.
2. **Queueing the Task:** The newly created `TCCTask` (as an `ITask` interface) is immediately placed into the `FTasks` queue using `QueueTask`. This operation is protected by the `Monitor` to ensure thread safety.
3. **Activating an Inactive Thread:** After queuing, `ComputeCore` attempts to find an inactive worker thread using `ActivateInactiveThread`. This method atomically retrieves an idle `TCCThread` from the `FInactiveThreads` stack and marks it as active, all under the protection of the `Monitor`.
4. **Signaling the Thread:** If an inactive thread is found, its internal `FSignal TEvent` is set (`thread.Signal.SetEvent`). This wakes up the waiting worker thread, signaling that there's work to be done.
5. **Task Execution:** If no inactive thread is immediately available, the task simply remains in the `FTasks` queue. It will be picked up by the next worker thread that finishes its current task and becomes available.

Worker Threads: The Engine of ComputeCore

The `TCCThread` objects are the tireless engines driving `ComputeCore`. These are persistent `TThread` descendants that form the core of the thread pool.

- **Simple Execution Loop:** Each `TCCThread` runs a straightforward `Execute` method loop:

  Code snippet

  ```
  while (not Terminated) and (FSignal.WaitFor <> wrTimeout) do begin
    while (not Terminated) and FOwner_ref.AllocateTask(Self, task) do
      task.Execute;
  end;
  ```

- **Waiting for Work:** If the task queue (`FTasks`) is empty, the worker thread calls `FSignal.WaitFor`. This puts the thread into an efficient waiting state, consuming minimal CPU cycles until its `FSignal` event is triggered by `TComputeCore.Run` when a new task is queued and an inactive thread is activated.[6] This prevents "busy-waiting".[6]
- **Processing Tasks:** When `FSignal` is set (or if there were tasks already in the queue when the thread checked), the thread enters an inner loop. It repeatedly calls `FOwner_ref.AllocateTask(Self, task)`. This method, which is thread-safe, attempts to retrieve a task from the queue. If a task is successfully retrieved, the worker thread immediately calls `task.Execute`.
- **Returning to Wait:** Once the inner loop finishes (meaning no more tasks are immediately available in the queue), the worker thread returns to its outer loop, where it will again wait on `FSignal` until new work is signaled.
- **Task Exception Handling:** Inside `TCCTask.Execute`, the `FTaskProc` (the user's task code) is wrapped in a `try..except` block. Any exception raised by the task is caught, and the exception object is stored in `FExceptObj`. This prevents unhandled exceptions in worker threads from crashing the entire application.[2] The

  `FWorkDone` event is always signaled in a `finally` block, ensuring completion notification regardless of success or failure.

Race Conditions: Anticipating and Mitigating Concurrency Bugs

A **race condition** occurs when the outcome of a program depends on the sequence or timing of uncontrollable events, such as the order in which multiple threads access and modify shared data. These are notoriously difficult to debug because they often manifest intermittently.[2]

`ComputeCore`'s design is centered on mitigating these issues.

- **Critical Parts:** In `ComputeCore`, the critical parts of the code are those that access and modify the shared data structures: the `FTasks` queue and the `FInactiveThreads` array.
- **The "Missed Task" Race Condition:** Consider what happens if a task is being scheduled (added to `FTasks`) at the exact same time as another task finishes execution and a worker thread becomes inactive. If not handled carefully, it could happen that a task is put into the internal task queue, but none of the threads would start working on it because the "check for

inactive thread" and "get task" operations are not atomic. A thread might become inactive *just after* the check, or a task might be added *just after* the queue is checked.

- `AllocateTask` **as the Solution:** This is precisely why `TComputeCore.AllocateTask` does more than one job from within the locked area. It atomically performs two crucial steps:

1. It tries to get a task to be processed (`GetTask_U`).
2. It then marks the calling worker thread either active or inactive based on whether a task was successfully retrieved (MarkThreadActive_U).

This entire operation is enclosed within __Acquire and __Release calls, ensuring it is an atomic unit. If this operation were split into two separate critical sections, it would create a possible race condition where a task could be queued, but no thread would be signaled to pick it up, leading to a stalled task.

Preventing Resource Exhaustion: A Balanced Approach

Resource exhaustion is a significant concern in multithreaded applications, manifesting as excessive CPU usage, memory leaks, or an unresponsive system. `ComputeCore` employs several strategies to prevent this, ensuring a balanced and efficient operation.

- **Thread Pooling:** By creating a fixed number of `TCCThread` objects at startup and reusing them, `ComputeCore` avoids the overhead and resource consumption associated with constantly creating and destroying threads.[1] This limits the total number of active threads, preventing the operating system from spending excessive time on context switching.
- **Efficient Waiting:** As discussed, worker threads use `TEvent` (`FSignal`) to wait efficiently when no tasks are available, preventing "busy-waiting" and conserving CPU resources.[6]
- Work-Stealing in WaitFor: This is a crucial mechanism to prevent deadlocks and resource exhaustion, especially in scenarios like recursive algorithms (e.g., QuickSort) where tasks might spawn nested tasks. If a task recursively spawns children and then waits for them, it's easy to exhaust all worker threads, as they would all be busy waiting for other tasks to complete, with no threads left to process the remaining child tasks.

TComputeCore.WaitFor addresses this by implementing a "work-stealing" approach:

Code snippet

```
while task.WorkDone.WaitFor(0) = wrTimeout do begin
  __Acquire;
  GetTask_U(newTask);
  __Release;
  if not assigned(newTask) then
    TThread.Yield // No tasks available, yield CPU
  else begin
    newTask.Execute; // Execute a stolen task
    if assigned(newTask.ExceptObj) then
      raise Exception(newTask.ExceptObj); // Re-raise exception from
stolen task
  end;
end;
```

While waiting for the specified `task` to complete (`task.WorkDone.WaitFor(0)`), the `WaitFor` method actively tries to "steal" and execute other tasks from the `FTasks` queue. If it finds a `newTask`, it executes it directly. If no tasks are available, it calls `TThread.Yield` to give up its time slice, allowing other threads to run. This ensures that even if a thread is blocked waiting for a child task, it can still contribute to processing other pending tasks, preventing a deadlock and keeping the pool productive.

Unit Testing

Let's talk about unit testing in a separate article.

Conclusion: The ComputeCore Advantage

`ComputeCore` stands as a testament to the power of well-designed, specialized units in Delphi development. Its clear goals, deliberate implementation choices, and robust handling of concurrency challenges make it a valuable asset.

Its core advantage lies in its ability to **simplify complex concurrency challenges**. By abstracting away the low-level details of thread management and offering a task-oriented API, `ComputeCore` allows developers to focus on their application's logic rather than getting entangled in the pitfalls of raw threading.[7] This simplification directly translates to

enhanced application responsiveness and improved performance, making full use of modern multi-core processors while keeping the user interface fluid and interactive.

Furthermore, `ComputeCore`'s adherence to principles like thread pooling, efficient synchronization using `Monitor` and `TEvent`, and the clever work-stealing mechanism in `WaitFor` ensures a high degree of **stability and reliability**. It inherently mitigates many common multithreading bugs, providing a safer environment for concurrent operations.

Ultimately, `ComputeCore` acts as a powerful "pre-built" foundation, much like the utility units often highlighted by thedelphigeek.com, empowering Delphi developers to build more sophisticated, responsive, and robust applications with greater ease and confidence. It's about making multithreading in Delphi not just possible, but genuinely practical and efficient.

It is not that hard to write your own parallel processing framework if you follow these rules:

- Keep it simple!
- Minimize access to shared data structures!
- Think about race conditions!
- Do lots of testing!