

# **Preemptive Processes on 4D Remote**

By Timothy Aaron Penner, Senior Technical Services Engineer, 4D Inc.

Technical Note 19-10

## Table of Contents

---

Table of Contents .....	2
Introduction .....	3
Overview .....	3
Difference between Preemptive and Cooperative modes.....	3
Requirements for using preemptive processes .....	4
The 64-bit version of 4D / 4D Server must be used .....	4
The application code must be Compiled.....	4
Each part of the code must be Thread-safe .....	4
Checking thread safety via the docs .....	4
Checking thread safety via the compiler .....	5
Using the New Network Layer (for 4D Remote) .....	6
Preemptive on 4D Remote .....	6
Preparing environment for preemptive .....	6
Preparing code for preemptive .....	7
What if the offending command cannot be removed from the method?.....	9
Calling cooperative tasks from preemptive process asynchronously.....	9
Calling cooperative tasks from preemptive process synchronously .....	10
Checking if the process is preemptive .....	10
Example .....	11
Conclusion .....	11

## Introduction

---

Multi-core computer systems are readily available, and customer expect applications to take advantage of the extra cores within their machines. Traditionally, 4D has been a single-threaded application – but overtime more and more aspects of the 4D application have been updated to take advantage of additional cores. Initially, the backend parts of 4D, such as the SQL and HTTP servers, were updated to take advantage of multi-cores. Later, language commands were progressively made thread safe. Originally this was only available in single-user mode or when the code was executed on the server. Now, starting with v17 R4, 4D Remote (i.e. 4D Clients) can also take advantage of multi-core systems as well. This tech note discusses the requirements for using preemptive mode as well as the steps necessary to ensure a thread-safe process.

## Overview

---

Before talking about what's new in v17R4, let's first recap some of the terms and requirements surrounding this feature.

### **Difference between Preemptive and Cooperative modes**

Cooperative mode was the only mode available in 4D until 4D v15 R5. Cooperative means that the 4D processes cooperate (work together) to share a single CPU. Sometimes they cooperate explicitly, like when the IDLE command is used, and sometimes they cooperate implicitly, like when most any other 4D language command is used. This is because, internally, 4D uses the IDLE command to give the CPU to another process. As a consequence, only one CPU core can be used at a time while executing cooperatively.

In preemptive mode, the IDLE command does not need to be called because the 4D processes don't need to cooperate in order to share the CPU: it's done by the operating system. The operating system preemptively gives the CPU to one thread or another. So, if using a machine with multiple CPUs, then several threads can be executed at the same time.

As a result, in preemptive mode, overall performance of the application is improved, especially on multi-core machines, since multiple processes (threads) can truly run simultaneously. However, actual performance gains depend on the operations being executed.

In return, since each thread is independent from the others in preemptive mode, and not managed directly by the application, there are certain limitations on what can be executed. We refer to this as thread-safe. In order to execute preemptively the process must only use thread safe commands. Additionally, preemptive execution is only available in certain specific contexts (i.e. 64-bit compiled with new network layer).

## Requirements for using preemptive processes

In order for 4D to be able to execute code preemptively, there are certain requirements that must be met. The following section outlines these requirements and explains how to meet them.

### The 64-bit version of 4D / 4D Server must be used

Preemptive mode is only available in 64-bit versions of 4D; the 32-bit versions of 4D are not compatible with the preemptive execution mode. Moving forward, 4D is only available in 64-bit editions (starting in v17R5) so this requirement should be fulfilled automatically.

### The application code must be Compiled

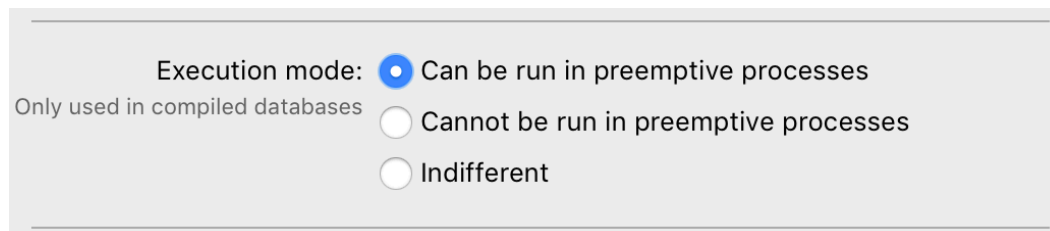
Preemptive execution is only available when 4D is running in Compiled context, which includes the following deployment options:

- Merged double-clickable applications
- Compiled 4DC structure files
- Interpreted 4DB structures running in **Compiled** context

Preemptive mode is **not** available when 4D is running in Interpreted context.

### Each part of the code must be Thread-safe

The entire process must be thread safe, including all commands and methods called from within the process. Starting with the method being passed to **CALL WORKER** or **New process** command, must have the **Execution Mode** set to “can be run in preemptive processes” within the **Method Properties**.



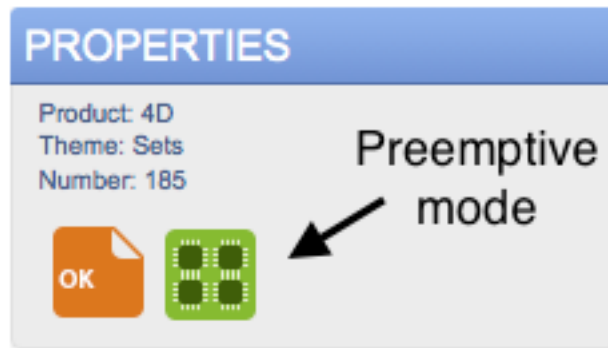
Execution mode:  Can be run in preemptive processes  
Only used in compiled databases  Cannot be run in preemptive processes  
 Indifferent

Every command called from within that method must also be thread safe. If the method calls other methods, then those methods must also be thread safe as well as every command within those methods.

Command thread safety can be checked in the documentation as well as by the compiler. 4D v17 includes thousands of thread safe commands, but some legacy commands (and commands dealing with the UI) are not thread safe.

### Checking thread safety via the docs

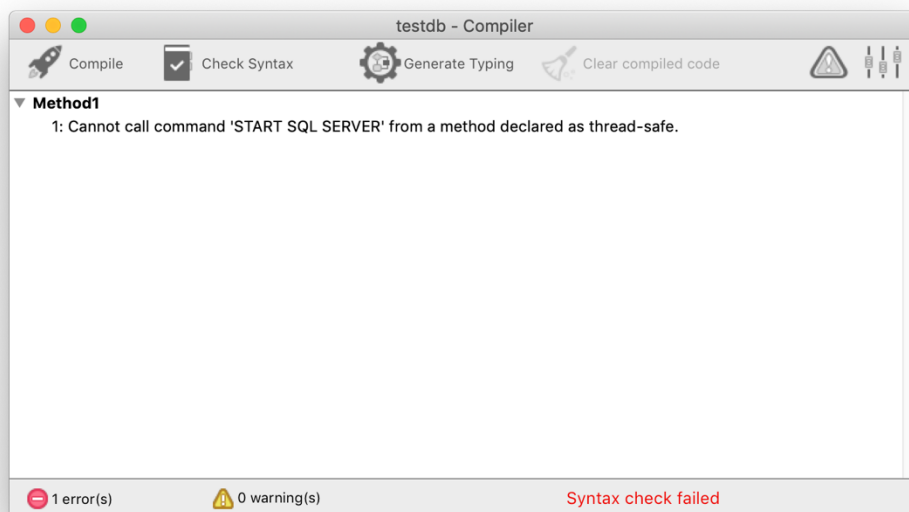
To check thread safety via the documentation, look for the Preemptive mode icon in the command Properties infobox, located on the right-hand side of the documentation page:



The documentation screenshot above shows the green preemptive icon, indicating that this command is thread-safe therefore it is OK to use in a preemptive method.

### Checking thread safety via the compiler

To check thread safety via the compiler, attempt to compile or check the syntax and look at the result to see if there are thread safety errors:



The compilation window above is showing an error, notifying the developer that the START SQL SERVER command is not thread safe.

Since this command is not thread safe the developer will have to re-architect the process so that it is no longer calling the command directly.

## Using the New Network Layer (for 4D Remote)

Starting with 4D v17R4 preemptive mode is available in 4D Remote in addition to being available in 4D Server and Standalone. In order to take advantage of the preemptive execution from a 4D Remote context the New Network Layer must be used for Client-Server communication.

The New Network Layer is enabled by default in new applications but upgraded applications may still have the legacy network layer

If “use legacy network layer” is enabled the 4D Remote (clients) will be unable to use preemptive mode.

## Preemptive on 4D Remote

---

Starting with v17R4, preemptive process can now run on 4D Remote. This section covers in greater detail the items needed for preparing the environment and the application.

### Preparing environment for preemptive

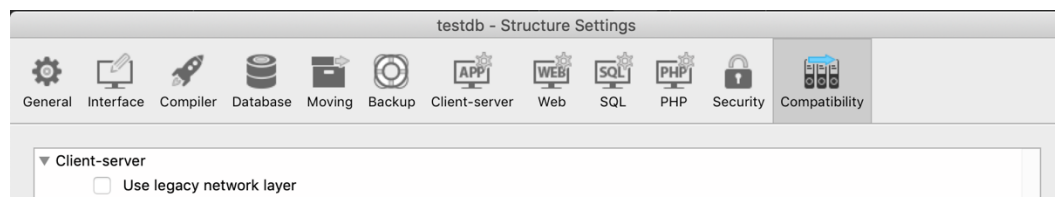
Preparing the environment is as easy as 1, 2, 3:

- Use v17R4 **or newer** (*preemptive execution on 4D Remote is a v18 feature*)
- Use 64-bit version of 4D
- Use the New Network Layer

The first step to preparing for preemptive mode on 4D Remote is to be sure to be using the appropriate versions of 4D. Preemptive execution on 4D Remote is a feature of v18 and was first made available in v17R4. Preemptive execution on 4D Remote is not available in v17.x (i.e. v17.0, v17.1, v17.2, and etc.). So, be sure that the appropriate version of 4D is used when attempting to use preemptive execution on 4D Remote.

The next step is to make sure that the 64-bit version of 4D is being used. Luckily, the 32-bit versions of 4D have been deprecated since v17.0 and the only versions available on the download page are 64-bit.

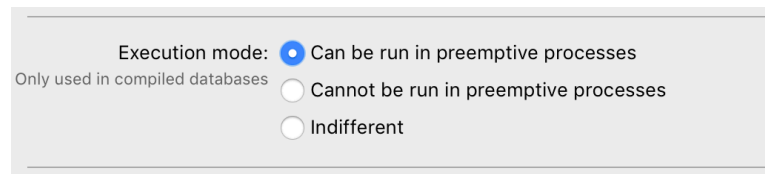
The final step in preparing the environment is to make sure that the New Network Layer is used. Check within the Database Settings -> Compatibility options to determine the current setting:



If “Use legacy network layer” is enabled, it must be disabled in order to use the new network layer.

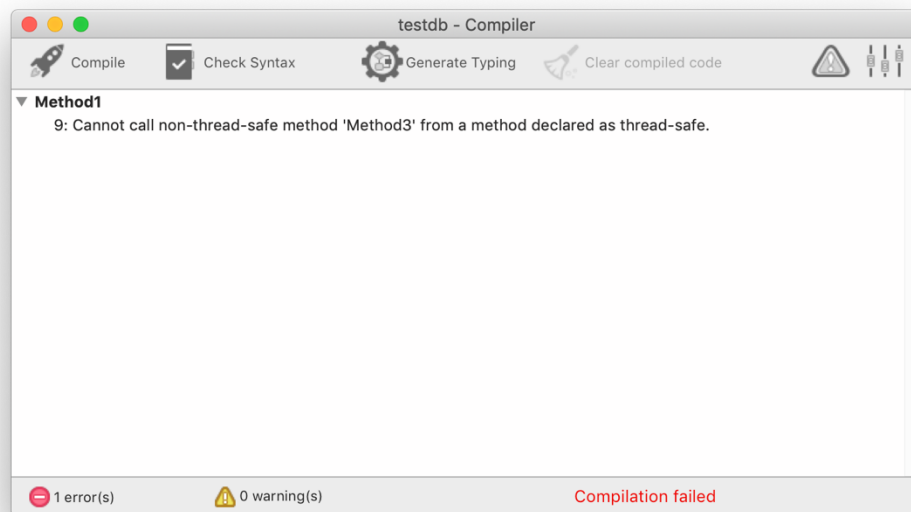
## Preparing code for preemptive

By default, 4D will execute all project methods in cooperative mode. If the developer wants to benefit from the preemptive mode feature, the first step consists of explicitly declaring all methods that they want to be preemptive within the method properties.



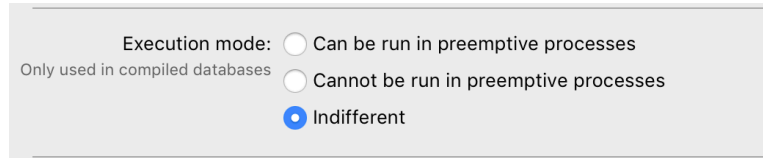
This needs to be done for each method that is to be run preemptive.

Next, compiling the code is required – if the method contains code that is not thread safe the compilation of the code will fail with an error like this:



The screenshot above is indicating that Method1 (which is marked as 'can be run in preemptive processes') is calling a method (Method3) that is not declared as thread safe. However, the compiler does not indicate what part of code is causing a problem, it simply states that the entire method (Method3) is not thread-safe.

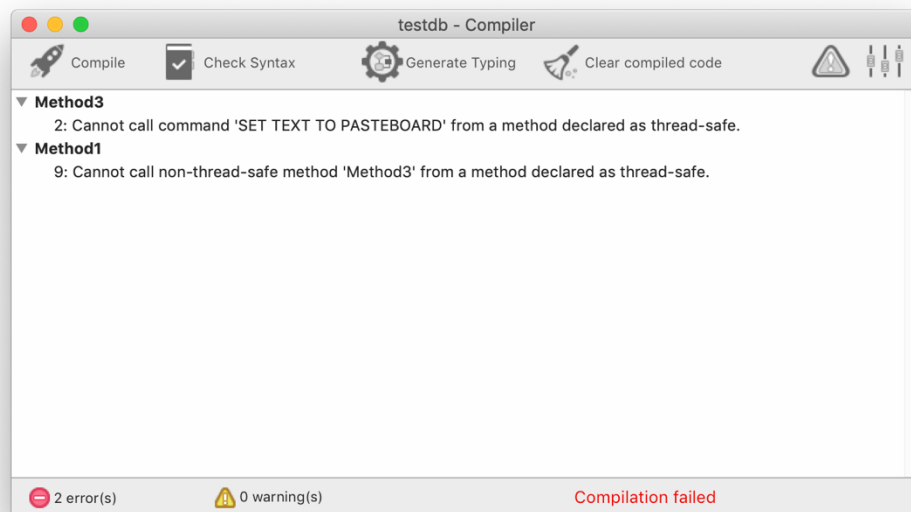
To correct this error, check the method properties for the method listed in the error (Method3). Upon checking the method properties for Method3 we see that the **execution mode** property is set to indifferent:



With this setting, the compiler is able to detect if the method is thread safe, but it doesn't check the contents to determine which piece of code is causing the problem.

To correct this situation and make the compiler check a bit deeper, the developer can declare in the method properties that the method "can be run in preemptive processes".

Upon changing the **execution mode** method property and re-running the compiler we are able to get additional clues about why the method was failing to pass the thread-safety check:

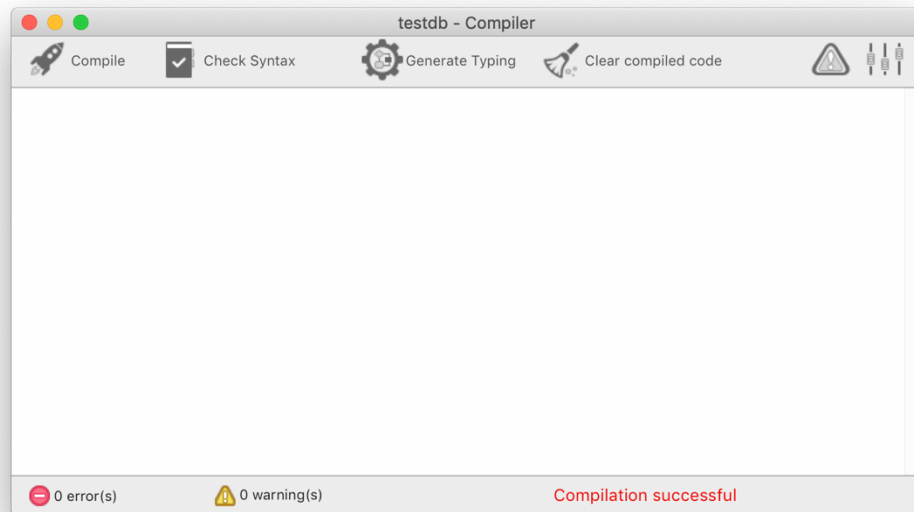


The screenshot above is indicating that the underlying error is due to the use of the **SET TEXT TO PASTEBOARD** command within the method.

To correct that situation, replace the functionality of the command with a thread safe command. In this specific situation we can simply remove the call to the **SET TEXT TO PASTEBOARD** command to make this method thread-safe.

After removing the offending command from the preemptive call chain, re-run the compilation check to see if it passes.





If compilation is successful, then the application can be relaunched for further testing in compiled mode.

### **What if the offending command cannot be removed from the method?**

The last example assumes that the command can simply be removed, but what if the code or logic cannot be removed from the method?

If the none-thread-safe code cannot be removed from the method, then the short answer is that this method cannot be run in preemptive context – but if the developer is determined there may be a way to refactor it.

The solution depends on whether or not the code needs to be run synchronously or if it could be run asynchronously.

### **Calling cooperative tasks from preemptive process asynchronously**

If a preemptive process needs to call a cooperative task and the preemptive process does not need to wait for the cooperative task to finish, then it could be called asynchronously via the **New process** command. This approach is very easy and simply consists of spinning off a new process to handle the cooperative task. This approach is very effective if the preemptive process does not require a result cooperative task and does not need to wait for the cooperative task to complete.

If the preemptive task must wait for the cooperative task to complete, then the developer would want to call the cooperative in a synchronous manner. In this way, the preemptive task would pause and wait for the cooperative task to complete. An approach for this is discussed in the next section of this document.

## Calling cooperative tasks from preemptive process synchronously

If a preemptive process needs to call a cooperative task and the preemptive process must also wait for this cooperative task to finish, then the developer could use the new `Signal` command as a way of pausing the preemptive process to wait for the other process to complete.

Start by calling the **New signal** command to get a *signal* object. A signal is a special type of shared object that can be passed as parameter from a worker or process to another worker or process, so that:

- the called worker/process can update the signal object after specific processing has completed
- the calling worker/process can stop its execution and wait until the signal is updated, without consuming any cpu resources.

This is particularly useful for allowing synchronous work between a preemptive and cooperative thread.

The signal object has the following properties attached to it that allow further interaction between the cooperative and preemptive tasks:

- `signal.signaled`
  - Boolean, read-only property. This is **false** initially and becomes **true** after the `signal.trigger()` method is called.
- `signal.description`
  - Text, this can contain a custom description of the signal (optional)

The signal object also has the following methods attached to it to help facilitate interaction between:

- `signal.wait()`
  - this is used by the calling process to wait
  - can be used with no value to wait indefinitely
  - can be used with a value to specify the number of seconds to wait
- `signal.trigger()`
  - this is used by the called process to trigger the signal and set `signal.signaled` to **true**.





Due to the nature of a shared object, the signal object can also be used to pass information back and forth between preemptive task and cooperative tasks.

## Checking if the process is preemptive

4D provides multiple ways of checking to see if a process is executing in preemptive mode or cooperative mode, include:

- The **PROCESS PROPERTIES** command allows you to find out whether a process is run in preemptive or cooperative mode.

- Both the Runtime Explorer and the 4D Server administration window display specific icons for preemptive processes:

Process type	Icon
Preemptive stored procedure	
Preemptive worker process	
Preemptive Web method	
Preemptive SOAP method	

## Example

The following method can be used to check if a process is executing in preemptive or cooperative contexts:

```
C_BOOLEAN($0)
C_TEXT($procName)
C_LONGINT($1;$procState;$procTime;$procMode)
PROCESS PROPERTIES($1;$procName;$procState;$procTime;$procMode)
$0:=$procMode ?? 1
```

The method above takes a process number as a parameter and returns true if the specified process is executing in preemptive mode, false if it is executing in cooperative mode.

## Conclusion

---

This technical note described the general concepts of using preemptive processes from a 4D Remote (client). Two techniques for refactoring none-thread-safe tasks from methods were presented. This information should allow the 4D developer to write and test preemptive code and choose whichever refactoring technique they prefer.