CPU, scheduler and processes

By Jean-Yves Fock-Hoon, QA Manager

Technical Note 05-20

Overview

This technical note will try to help you to understand how 4D works, therefore, helping you how to take advantage of the current architecture to speed up your database.

More speed

The idea is far from new. Any application will run faster on a faster machine. Getting more processing power for a given machine is the idea we will explore here from a software standpoint. Processes have different levels of priorities. Each process will use the CPU time assigned by the system and may release the CPU time in order to allow other processes to have their share of CPU time to run.

There is a setting in the Database properties where you can set the CPU usage in 4D. You are given the choice to increase or decrease the CPU time will be using on the machine. The Preference dialog allows you to choose between three settings: Low, Normal or High. The default value is Normal and is suitable for most applications. Low allocates less CPU time to 4D and High allocates more. If those three settings do not fit your needs, you can define custom values using the SET DATABASE PARAMETER command (selectors 10, 11 and 12).

The selector 10 (`4th Dimension Scheduler`) applies the setting to your 4D stand alone application.
The selector 11 (`4D Server Scheduler`) applies the setting to your 4D Server application.
The selector 12 (`4D Client Scheduler`) applies the setting to your 4D Client application.

These settings are defined for the current application and can be changed at any time. You can assign a new value by using the SET DATABASE PARAMETER command (http://www.4d.com/docs/CMU/CMU00642.HTM). To retrieve the current value, use the Get database parameter command (http://www.4d.com/docs/CMU/CMU00643.HTM)

Here is an example of code that you can use to retrieve the current value:

$lparams:=**Get database parameter** (4th Dimension Scheduler)

```
vNbTicks:=$lparams & 0x00FF
vMaxTicks:=($lparams >> 8) & 0x00FF
vMinTicks:=($lparams >> 16) & 0x00FF
```

Here is an example of code that you can use to compute a new value:

```
$lparams:= vNbTicks +( vMaxTicks << 8)+( vMinTicks << 16)
SET DATABASE PARAMETER (4th Dimension Scheduler; $lparams)
```

vMinTicks and vMaxTicks are the minimum and maximum number of ticks per call to the system. When the system allocates CPU time to 4D, 4D is bound to release it during a subsequent call to the OS. The longer the call to the OS, the more likely 4D is to release the CPU time. This means that 4D will be slower since most of the CPU time will be used by the operating system. The operating system may reassign the CPU time to other applications or it may go to waste since no application is using it.

Lower values correspond to higher CPU time allocated to 4D. However this may slow down the OS or other applications that are running on that machine. With 4D Client, you may also increase the number of lost connections, because the OS is starved from CPU time. Increasing the CPU share on 4D client will also improve performances with slow network configurations.

A big gap between the minimum and maximum values would mean that 4D will use the whole CPU when offered, but is ready to share with the system if needed. This means that 4D should be fast but may become slower if other applications require more CPU.
NbTicks is the number of ticks that 4D wait for before checking if it should release the CPU time.

Settings provided by default within 4D are usually enough and it is rare to have to use custom values. It is recommended to keep these default settings. However, if you choose to use different settings, we recommend that you stress-test your application to make sure the resource availability is acceptable for 4D, the OS and other applications.

The Scheduler

4D is a thread that has its own scheduler to emulate processes within 4D. This means that when 4D receive CPU time from the OS, 4D needs to spend some CPU time in that scheduler and allocate some CPU time to its own processes. The scheduler will also release the CPU time to the system to share the CPU with other applications. These 4D processes would typically execute some 4D code. The problem is that a 4D process may require a lot of

CPU time when executing a specific command or method and you would need to avoid the monopolization of the CPU by one process since it means that other 4D processes won't be able to work (this can generate some disconnections in Client/Server configuration too).

It also means that the scheduler won't be able to call back the system and other applications running on the same machine would also suffer this lack of CPU time. This is why a 4D process always needs to give the hand back the to the 4D scheduler as soon as possible in order to avoid such issues.

When running in interpreted mode, 4D processes will always call back the 4D scheduler when executing any command or methods. At this point, the 4D scheduler may give back the hand to 4D processes or to the system.
If you compile the database, it's because you want the code to be faster. In that case, a callback may not be performed since it could slow down your process, which is not what you want. On the other hand, if your processes never perform callbacks, it would mean that you would never have more than one 4D process since you would never be able to dedicate some CPU time to other 4D processes. This is why it's sometimes important to perform callbacks in your 4D processes.

This is why a callback is performed after executing any major commands. But performing too many callbacks slows down your 4D process. This is why 4D will not perform callbacks when executing very fast operations, such as calculations, functions or when looping (For, While…) for example.
If you write the following code:

```
For ($i;1;200000)
  $MyVar:=4
End for
```

When running that code in compiled mode, your process will never perform any callback in that loop. This is why you might need to add the IDLE command. This command will give back the hand to the 4D scheduler that will redistribute the CPU time as explained above.

Idling in 4D

While most major 4D commands will perform a callback, others will not; this is why we need to execute sometimes the IDLE command. In fact, calling IDLE may or may not perform a callback in 4D. These major commands and the IDLE command are in fact, requesting a callback. At that point, the scheduler makes sure that each process uses at least one tick.
This is why executing 10 IDLE in a row can be useless. These 10 IDLE may be executed in less than 1 tick. Each time that an "Idle" is performed, the scheduler will see that the process which used less than 1 tick calls back the calling process. Our process will execute the second IDLE, call back the

scheduler only to be called almost right away and so on. We can say that, instead of idling, we're just wasting CPU time. If you really want your process to not use this 1 tick, you may execute the DELAY PROCESS (0) command. This forces a callback to the scheduler.

So, if you are executing multiple processes, you need to be sure that all of them will have a fair share of CPU time. You may avoid to 'overuse' IDLE or DELAY PROCESS (0). This would decrease the CPU time allocated to your process and may affect performance.

Processes

As we saw, the scheduler will assign CPU time to 4D processes. It's up to the code in the 4D processes to ensure that we do not spend too much time in them. However, creating and deleting a lot of processes is very expensive in CPU time, especially in a Client/Server configuration.

Usually, when someone no longer needs a process, instead of deleting the process, he just hides that process. CPU-wise, this makes no difference. The scheduler will still assign at least one tick to that process. In fact, after hiding the process, you should perform a PAUSE PROCESS for that process. When you do that, the scheduler will not assign any CPU time to the process. You may also perform a DELAY PROCESS on that process too. When you specify a DELAY PROCESS of 200 ticks for a process, the scheduler will not assign any CPU time to that process for the duration of the delay. DELAY PROCESS will act as PAUSE PROCESS, except that with DELAY PROCESS, the 4D scheduler will not assign CPU time to that process until the specified time has been reached.

When you no longer need a process and you know that you may have to recreate that process later, you may just move that process into a 'process pool', a pool of unused process and then, pause that process. If you need a process, just grab one process in that pool and resume it. If the pool is empty, you may then create a new process. This is the technique that 4D Client or some 4D developers are using when managing the web server.

Each time you want to create a process, 4D needs to allocate memory to that process (variables) and add that process to the current list (process information). This costs a lot of CPU, especially when variables tables are very big, and even more expensive with slow network connections in Client/Server configurations. This idea of pool can be interesting, if your client application creates a lot of processes that do not last long, you may save time by re-using them rather than deleting them (cleaning the memory) and recreating them (initializing the memory).

However, 4D Server and 4D Client have also their own 4D scheduler. When a process on 4D client performs a request to 4D Server, the Server needs to send back the answer to the client. The Server is going to 'stack' the request. The 4D Server scheduler will assign some CPU time to each process that will grab the incoming request, and provide a response. The Server responds to the Client. The Client is

going to then 'stack' the response. When the 4D Client scheduler gives CPU time to the process that made the request, it grabs the response and analyzes it. If any scheduler fails to assign some CPU time to these processes, an internal time-out error may occurs and then both Client and Server may remove the process from their list. And when the scheduler finally assigns some CPU time to that process, another timeout will be generated because there is no response from the other application.
This is why you can see some -10001 or -10002 errors with 4D Client.
- 4D Client can be so busy that it does not respond to any request from 4D Server. At this point, you can see the client disappearing from the list of processes on the Server side. When the Client responds, at last, it's too late. A network error is displayed.
- 4D Server can be so busy that it does not acknowledge to any request from 4D Client. 4D Client believes that 4D Server is no longer available and will display the -10002 error message.

This is why it's also important to not to have too many processes on the Server side too. If your processes will not deal with the data file, you might consider defining them as local processes. This will decrease the number of processes on 4D Server. The scheduler on 4D Server will be more efficient.

Memory allocation

When deleting a process, 4D needs to free the memory used by that process. When creating a new process, 4D needs to allocate the memory block to that process. This can be time consuming. This is why a pool of processes may help you to save some time. But keeping a pool of processes also use memory. This is why it is also important to reduce your variable tables, keep it them small, having few objects, removing unused objects. But this is not the only major issue.

People have tendencies to index all fields. The problem is that if multiple processes are trying to modify or delete records, this means that 4D also needs to update all index tables. 4D is forced to load and unload these index tables. Modifying a value means that 4D needs to remove the old key and insert a new one. Deleting records implies that you need to remove the key from that table. If you have an average of 50 indexed fields per table to be updated, with 50 tables and 100 processes, just imagine the workload that it implies.

Do not 'over-index' your fields. Updating your tables under heavy load may cost you a lot of CPU time. Index what need to be indexed only. Check out your triggers. They can also significantly slow down your applications. Enable only the event that will be triggered and keep them short to avoid 'bottlenecks'. Complex triggers are more likely to slow down your code. When they are very complex and long, you should execute the code in your methods and forms and keep the triggers for the database rules that apply directly to saving/loading of the data. Code that is executed outside triggers has less impact on other processes.

In summary

When it’s about to optimize performances in your database, it is important to understand how 4D uses the CPU time on the machine and how it assigns it to its own processes. You can act at two levels: by altering the CPU time 4D is getting or by fine-tuning how your processes are handled.