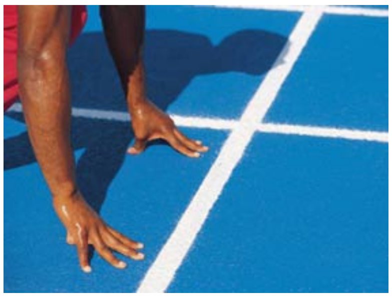
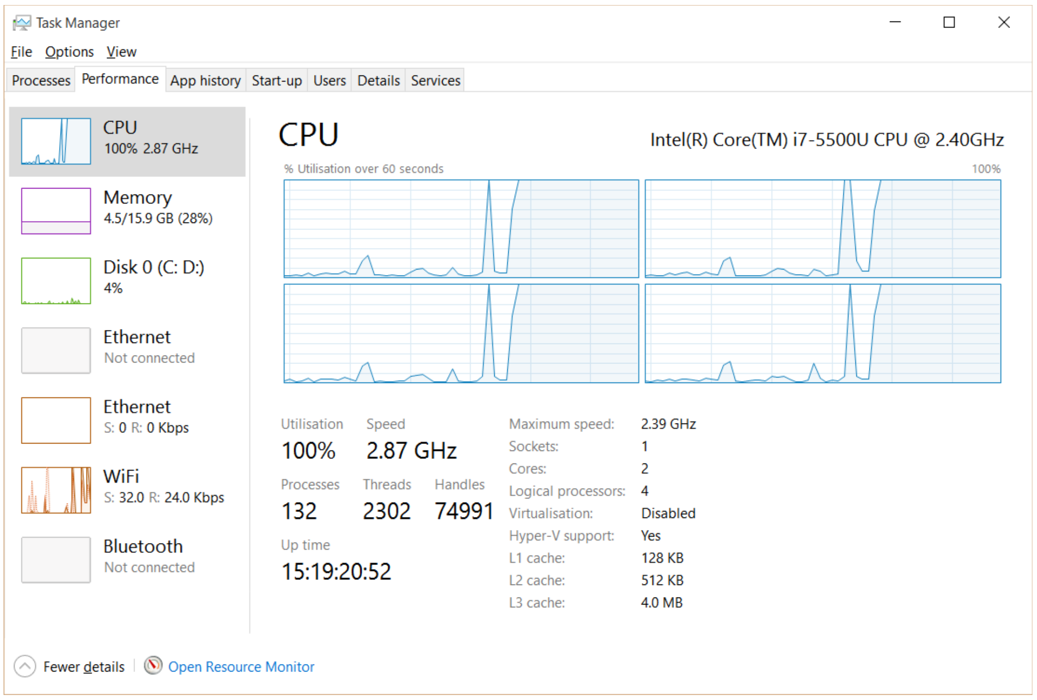


# maXbox Starter 42

Start with Multiprocessing Programming

## 1.1 Set a Core

Today we spend another small time in programming with multiple cores as multi-threaded programming. We will concentrate on one single function creating an API call of `ExecuteMultiProcessor`() pre-compiled in maXbox.



Task Manager

File Options View

Processes | Performance | App history | Start-up | Users | Details | Services

CPU 100% 2.87 GHz
Memory 4.5/15.9 GB (28%)
Disk 0 (C: D:) 4%
Ethernet Not connected
Ethernet S: 0 R: 0 Kbps
WiFi S: 32.0 R: 24.0 Kbps
Bluetooth Not connected

CPU Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz

% Utilisation over 60 seconds 100%

| Utilisation | Speed | | Maximum speed: | 2.39 GHz |
|---|---|---|---|---|
| 100% | 2.87 GHz | | Sockets: | 1 |
| Processes | Threads | Handles | Cores: | 2 |
| 132 | 2302 | 74991 | Logical processors: | 4 |
| | | | Virtualisation: | Disabled |
| Up time | | | Hyper-V support: | Yes |
| 15:19:20:52 | | | L1 cache: | 128 KB |
| | | | L2 cache: | 512 KB |
| | | | L3 cache: | 4.0 MB |

Fewer details | Open Resource Monitor

Multi-processing has the opposite benefits to multithreading. Since processes are insulated from each other by the OS, an error in one process cannot bring down another process. Contrast this with multi-threading, in which an error in one thread can bring down all the threads in the process. Further, individual processes may run as different users and have different permissions.

A multi-core processor is an integrated circuit (IC) to which two or more processors have been attached for enhanced performance, reduced power consumption, and more efficient simultaneous processing of multiple tasks like in parallel processing. A dual core set-up is somewhat comparable to having multiple, separate processors installed in the same computer, but because the two processors are actually plugged into the same socket, the connection between them is faster.

But, first of all I'll explain you what "blocking" and "non-blocking" calls are (later on more). In fact there are 2 programming models used in event driven applications:

- Synchronous or blocking
- Asynchronous or non blocking

Don’t confuse it with parallel programming (simultaneous) in which we can have two synchronous functions in dependence but each in a separate thread to split and speed up the results.
Sure, two asynchronous functions can be started at the same time if they can handle parallel core processing and that's what we do now:

```
 if ExecuteProcess(exepath+'maxbox3.exe '+
                  FILETO_RUN +' para1', SW_SHOW, 1, false) = 0 then
                    writeln('Multiprocessing Runs on CPU 1');

 if ExecuteProcess(exepath+'maxbox3.exe '+
                  FILETO_RUN +' para2', SW_SHOW, 2, false) = 0 then
                    writeln('Multiprocessing Runs on CPU 2');

 if ExecuteProcess(exepath+'maxbox3.exe '+
                  FILETO_RUN +' para3', SW_SHOW, 4, false) = 0 then
                    writeln('Multiprocessing Runs on CPU 3');

 if ExecuteMultiProcessor(exepath+'maxbox3.exe '+
                  FILETO_RUN +' para4', SW_SHOW, 8, false) = 0 then begin
                    writeln('Multiprocessing Runs on CPU 4');
                    ShowMessage(SysErrorMessage(GetLastError))
                  end;
```

Its just a name convention that the last of the 4 calls is `ExecuteMultiProcessor` to tell me its the last one in async mode but concerning operation its no difference between `ExecuteMultiProcessor` or `ExecuteProcess:`

```
function ExecuteProcess(FileName: string; Visibility: Integer;
                        BitMask: Integer; Synch: Boolean): Longword;
```

The function runs a program on a specified set of CPUs on a multiprocessor system! With the filename we specify the name of the program we want to launch, also with parameters:

```
Const FILETO_RUN ='examples/044_queens_performer3.txt';
```

The meaning of: exepath+'maxbox3.exe '+ FILETO_RUN +' para1', :

You can put some command-line parameters to the program to differentiate some operations:

```
  if ParamStr(2) = 'para1' then begin
    NB1:= 10;
    FILESAVE:= Exepath +'examples\ChessSolution_Res10_1codes.txt';
  end;
```

The `ParamStr` function returns one of the parameters from the command line used to invoke the current program and the `ParamIndex` parameter determines which parameter is returned. The related `FindCmdLineSwitch` function can be used to check for presence of parameters, as starting with a control character, such as - or /.

Second parameter of the main function is the `BitMask` which specifies the set of CPUs on which we want to run the program; the `BitMask` is built in the following manner:

```
   //bitmask ---> 1 means on first CPU, sync or async possible!
   //bitmask ---> 2 means on second CPU, sync or async possible!
   //bitmask ---> 4 means on third CPU, sync or async possible!
   //bitmask ---> 8 means on fourth CPU, sync or async possible - true-false!
```

For example: I have 4 processor cores and I want to run the specified process only on the CPU 2 and 4:
The corresponding bit-mask will be:

```
    1010 -->2^0 * 0 + 2^1 * 1 + 2^2 * 0 + 2^3 * 1 = 2 + 8 = 10
```

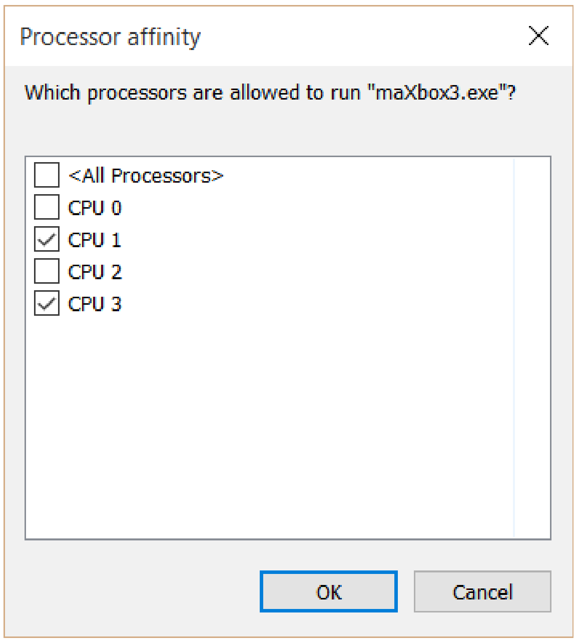
so the `BitMask` is 10 that you have to pass the function.

We call the 4 processes all simultaneous in asynchronous manner hence the master process isn't blocked.
When running on a single core its likely to be about one-and-a-half times slower than with multi core processor so the 4 cores need only twice the time as the single core.

```
start: 14:30:26:104
Solutions: 724   -ASCIITest: @ Filesize: 120856 KB
stop: 14:30:30:645
0 h runtime: 00:04:542 single core

Multi Core Solutions: 724   -ASCIITest: @

core 1 runtime: 00:09:196          core 2 runtime: 00:09:491
core 3 runtime: 00:08:257          core 4 runtime: 00:08:318
```

Ideally, a dual or more core processor is nearly twice as powerful as a single core processor. In practice, performance gains are said to be about fifty percent: a dual core processor is likely to be about one-and-a-half times as powerful as a single core processor depends on the payload.

Interesting is the dispatcher of the cores. Open the Task-manager, select the launched process and the right click, with "Set affinity", you will see a check on the CPUs you selected!


Processor affinity

Which processors are allowed to run "maXbox3.exe"?

- [ ] <All Processors>
- [ ] CPU 0
- [✓] CPU 1
- [ ] CPU 2
- [✓] CPU 3

In the main function we use the API call:

```
  //running process on the set of CPUs specified by BitMask
   SetProcessAffinityMask(ProcessInfo.hProcess, BitMask);
```

It sets a processor affinity mask for the threads of the specified process. If the function succeeds, the return value is nonzero.  The main functions exit code of the launched process (0 if the process returned no error ) is zero for success!

A process affinity mask is a bit vector in which each bit represents a logical processor on which the threads of the process are allowed to run. The value of the process affinity mask must be a subset of the system affinity mask values obtained by the `GetProcessAffinityMask` function. A process is only allowed to run on the processors configured into a system. Therefore, the process affinity mask cannot specify a 1 bit for a processor when the system affinity mask specifies a 0 bit for that processor.

Hope you did already work with the Starter 1 to 41 available at:

https://bitbucket.org/max_kleiner/maxbox3/wiki/maXbox%20Tutorials

So non blocking means that the application will not be blocked when the application plays a sound or a socket read/write data. This is efficient, because your application don't have to wait for a sound result or a connection. Unfortunately, using this technique is little complicated to develop a protocol. If your protocol has many commands or calls in a sequence, your code will be very unintelligible.

☞A function which returns no data (has no result value) can always be called asynchronously, cause we don't' have to wait for a result or there's no need to synchronise the functions.

Let's begin with the application structure process in general of an API call:
The `PlaySound` function plays a sound specified by the given file name, resource, or system event. (A system event may be associated with a sound in the registry or in the WIN.INI file.)

1. Header: Declared in `msystem.h`; include `Windows.h`.
2. Library: Use `Winmm.dll` or the lib.
3. Declaration of the external function
4. Load the DLL and call the API function
    - Static at start time
    - Dynamic at run time
5. Unload the DLL (loaded DLL: `C:\WINDOWS\system32\winmm.dll`)

So we call the API function dynamic at runtime and of course asynchronously. The sound is played asynchronously and `PlaySound` returns immediately after beginning the sound. To terminate an asynchronously played waveform sound, call `PlaySound` with `pszSound` set to NULL.

The API call to the function just works fine, doesn't have any glitch.

☞On the other side a sound is played synchronously, and `PlaySound` returns after the sound event completes. This is the default behaviour in case you have a list of songs.
If you click on the menu <`Debug/Modules Count`> you can see all the libraries (modules) loaded by your app and of course also the `winmm.dll` with our function `PlaySound()` in it.

```
25 function BOOL PlaySound(
  LPCTSTR pszSound,
  HMODULE hmod,
  DWORD fdwSound
);
```

## 1.2 Code with Cores

Before this starter code will work you will need to download maXbox from the website. It can be get from http://sourceforge.net/projects/maxbox site. Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox3.exe` the box opens a default program. Make sure the version is at least 3.9 because `Modules Count` use that. Test it with F2 / F9 or press **Compile** and you should hear a sound a browser will open. So far so good now we'll open the example:

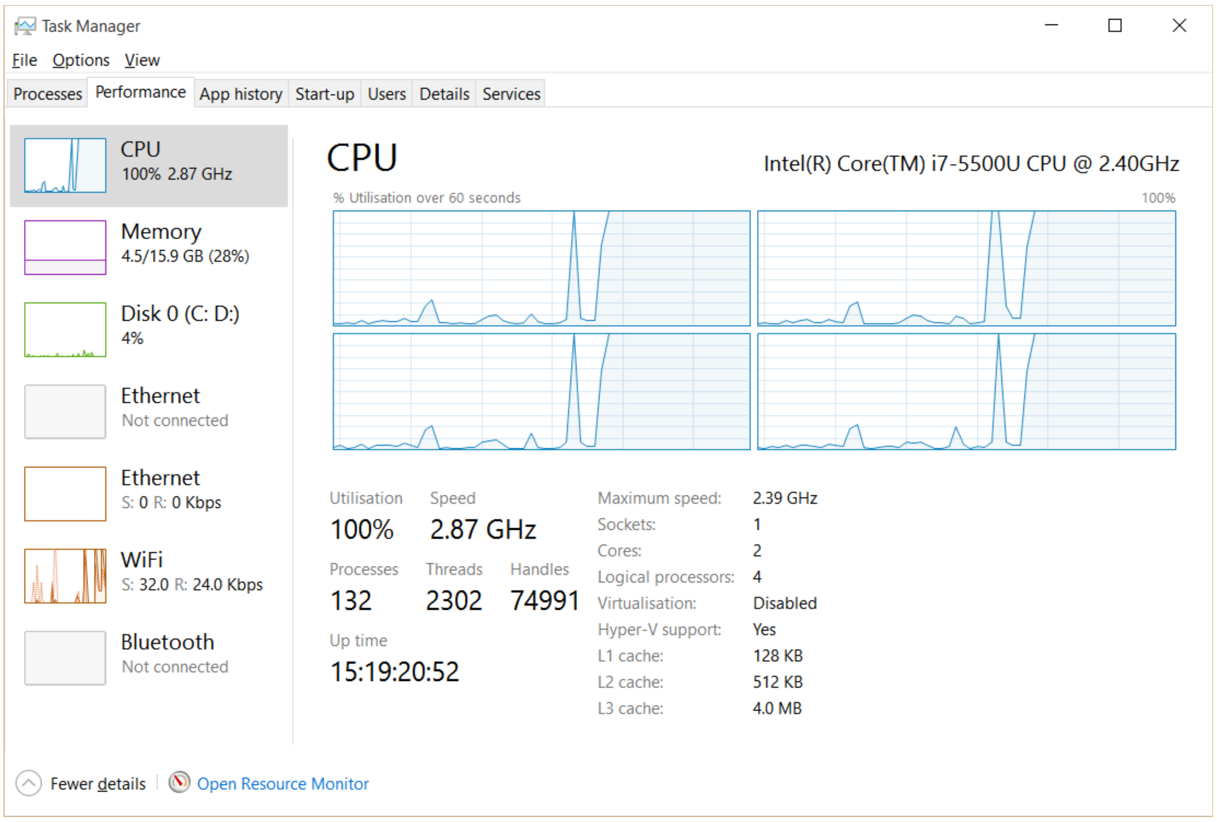
`630_multikernel3.TXT`

If you can’t find files use the link (size 23 KB):

http://www.softwareschule.ch/examples/630_multikernel3.TXT

http://www.softwareschule.ch/examples/263_async_sound.txt

Or you use the `Save Page as…` function of your browser[1] and load it from `examples` (or wherever you stored it). One important thing: The mentioned DLL `kernel32.dll` must reside on your disk (just a joke) and you should have at least a dual core processor. The main function is explained at the end of `630_multikernel3.TXT`.



Utilisation 100%
Speed 2.87 GHz
Processes 132
Threads 2302
Handles 74991
Up time 15:19:20:52

| | |
|---|---|
| Maximum speed: | 2.39 GHz |
| Sockets: | 1 |
| Cores: | 2 |
| Logical processors: | 4 |
| Virtualisation: | Disabled |
| Hyper-V support: | Yes |
| L1 cache: | 128 KB |
| L2 cache: | 512 KB |
| L3 cache: | 4.0 MB |

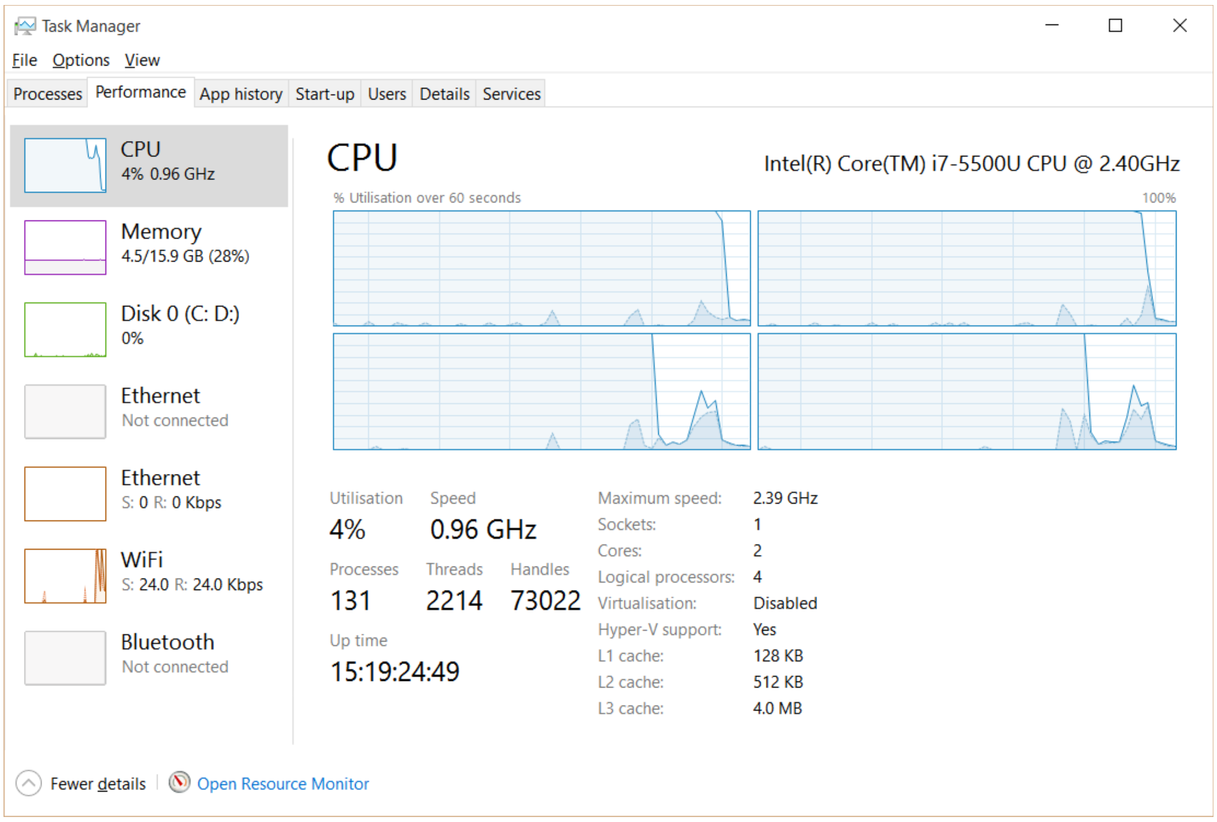
✌As we now know, `PlaySound` can't play two sounds at the same time, even if you do use `async` flag. You might be able to get this to work by having two separate threads both calling `PlaySound` synchronously.
The object will not let you play two sounds at once, even if you create two instances. You will need to bring in the native windows API "`mciSendString`".
You can test it with F4 or menu `/Output/New Instance` which opens a second instance of maXbox (see the following screenshot).
An example of the low-level `mciSendString()`:

```
mciSendString(@"open C:\Users\Desktop\applause.wav type waveaudio
alias applause", null, 0, IntPtr.Zero);
mciSendString(@"play applause", null, 0, IntPtr.Zero);
```

[1] Or copy & paste

```
mciSendString(@"open C:\Users\Desktop\foghorn.wav type waveaudio
alias foghorn", null, 0, IntPtr.Zero);
mciSendString(@"play foghorn", null, 0, IntPtr.Zero);
```

So your code will now show all three possibilities in a sequence. First with start with a dialog which stops the control flow because it's a modal dialog we have to wait and pass it. Second on line 28 our function is called in sync-mode `sync` and we have to wait or in other words we are blocked. Third in line 30 we call the same function with the `async` flag set to 1 and now you can follow at the same time the music plays and a loop of numbers on the output can be seen.
In the end of the multi-core processing you see also CPU time decay:



CPU Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz

Utilisation 4%
Speed 0.96 GHz
Processes 131
Threads 2214
Handles 73022
Up time 15:19:24:49

| | |
|---|---|
| Maximum speed: | 2.39 GHz |
| Sockets: | 1 |
| Cores: | 2 |
| Logical processors: | 4 |
| Virtualisation: | Disabled |
| Hyper-V support: | Yes |
| L1 cache: | 128 KB |
| L2 cache: | 512 KB |
| L3 cache: | 4.0 MB |

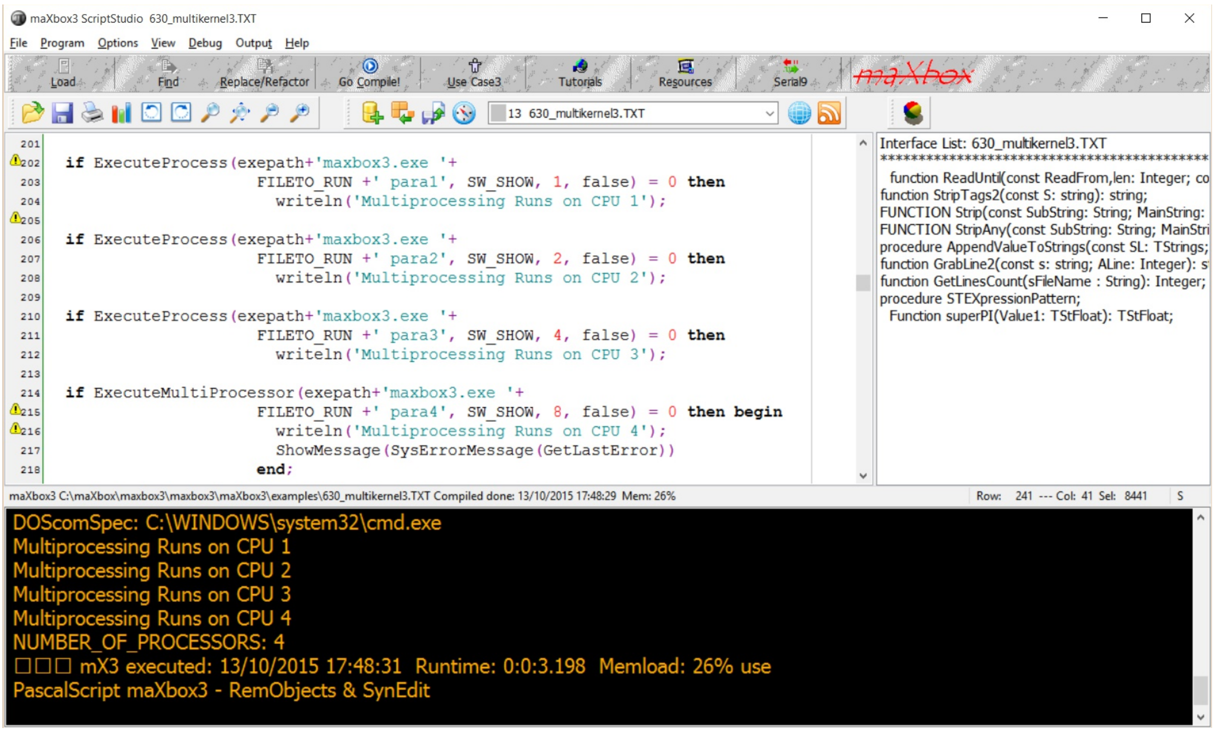
In line 33 we start almost the same time a sound twice, yes it's the same sound therefore you can hear the delay or an echo to prove its parallel! OK. It's a trick to open another function with the same song to show the simultaneous mode.
If you increase the delay of sound (for example with sleep or with a massive CPU payload), the echo effect space grows to more than milliseconds time of its start offset.

```
//Result:= Writeln(FloatToStr(IntPower(62,8))) = 218340105584896
27   ShowMessage('the boX rocks ' + MYS)
28   PlaySound(pchar(ExePath+'examples\maxbox.wav'), 0, 0);  //Sync
29   Sleep(20)
30   PlaySound(pchar(ExePath+'examples\maxbox.wav'), 0, 1);  //Async
31   //Parallel
32   //sleep(20)
33   closeMP3;
34   playMP3(mp3path);                                       //Parallel Trick
```

```
  for i:= 1 to 2300 do
    writeln(intToStr(i) + ' Aloha from ObjectPascal Bit');
  sleep(1250)
  inFrm.color:= clblue;
  //inFrm.close;
End.
```

☞This `sleep` is here just so you can distinguish the two sounds playing simultaneously.

By the way: You can also set a hash function around a sound file to distinguish it. A hash function is a (mathematical) function which receives an input of arbitrary length and returns a function value (hash val) of a fixed length (usually 128/160 bits).


maXbox3 ScriptStudio 630_multikernel3.TXT
File Program Options View Debug Output Help
Load Find Replace/Refactor Go Compile! Use Case3 Tutorials Resources Serial9
maXbox
13 630_multikernel3.TXT

```
 if ExecuteProcess(exepath+'maxbox3.exe '+
                     FILETO_RUN +' para1', SW_SHOW, 1, false) = 0 then
                       writeln('Multiprocessing Runs on CPU 1');

 if ExecuteProcess(exepath+'maxbox3.exe '+
                     FILETO_RUN +' para2', SW_SHOW, 2, false) = 0 then
                       writeln('Multiprocessing Runs on CPU 2');

 if ExecuteProcess(exepath+'maxbox3.exe '+
                     FILETO_RUN +' para3', SW_SHOW, 4, false) = 0 then
                       writeln('Multiprocessing Runs on CPU 3');

 if ExecuteMultiProcessor(exepath+'maxbox3.exe '+
                     FILETO_RUN +' para4', SW_SHOW, 8, false) = 0 then begin
                       writeln('Multiprocessing Runs on CPU 4');
                       ShowMessage(SysErrorMessage(GetLastError))
                     end;
```

```
Interface List: 630_multikernel3.TXT
*********************************************
 function ReadUntil(const ReadFrom,len: Integer; co
function StripTags2(const S: string): string;
FUNCTION Strip(const SubString: String; MainString:
FUNCTION StripAny(const SubString: String; MainStri
procedure AppendValueToStrings(const SL: TStrings;
function GrabLine2(const s: string; ALine: Integer): s
function GetLinesCount(sFileName : String): Integer;
procedure STExpressionPattern;
 Function superPI(Value1: TStFloat): TStFloat;
```

```
maXbox3 C:\maXbox\maxbox3\maxbox3\maXbox3\examples\630_multikernel3.TXT Compiled done: 13/10/2015 17:48:29 Mem: 26%    Row: 241 --- Col: 41 Sel: 8441    S
```

```
DOScomSpec: C:\WINDOWS\system32\cmd.exe
Multiprocessing Runs on CPU 1
Multiprocessing Runs on CPU 2
Multiprocessing Runs on CPU 3
Multiprocessing Runs on CPU 4
NUMBER_OF_PROCESSORS: 4
□□□ mX3 executed: 13/10/2015 17:48:31  Runtime: 0:0:3.198  Memload: 26% use
PascalScript maXbox3 - RemObjects & SynEdit
```

Till now we are discussing the topics of `sync` and `async` calls and the way it processes the calls by the receiver parallel or not. Asynchronous calls or connections allow your app to continue processing without waiting for the process (function) to be completely closed but not always in a simultaneous mode.

## 1.3 Notes about Threads

Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by:

- Avoiding bottlenecks. With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia

content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- Organizing program behaviour. Often, a program's run can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases.
- Multiprocessing. If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

One word concerning threads: Internal architecture name 2 thread categories.

- Threads with synchronisation (blocking at the end)
- Threads without synchronisation (non blocking at all)

In case you're new to the concept, a thread is basically a very beneficial alternative (in most cases) to spawning a new process. Programs you use every day make great use of threads whether you know it or not but you have to program it.
The most basic example is whenever you're using a program that has a lengthy task to achieve (say, downloading a file or backing up a database), the program (on the server) will most likely spawn a thread to do that job.
This is due to the fact that if a thread was not created, the main thread (where your main function is running) would be stuck waiting for the event to complete, and the screen would freeze.
For example first is a listener thread that "listens" and waits for a connection. So we don't have to worry about threads, the built in thread will be served by for example Indy though parameter:

```
   IdTCPServer1Execute(AThread: TIdPeerThread)
```

When our DWS-client is connected, these threads transfer all the communication operations to another thread. This technique is very efficient because your client application will be able to connect any time, even if there are many different connections to the server. The second command "`CTR_FILE`" transfers the app to the client:

Or another example is AES cryptography which is used to exchange encrypted data with other users in a parallel way but not a parallel function. It does not contain functions for key management. The keys have to be exchanged between the users on a secure parallel channel. In our case we just use a secure password!

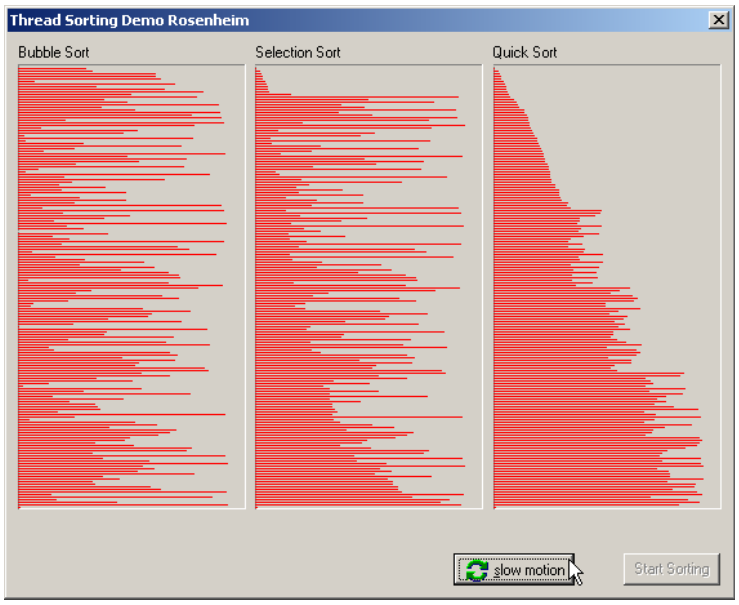
```
procedure EncryptMediaAES(sender: TObject);


 with TStopwatch.Create do begin
    Start;
    AESSymetricExecute(selectFile, selectFile+'_encrypt',AESpassw);
    mpan.font.color:= clblue;
    mpan.font.size:= 30;
    mpan.caption:= 'File Encrypted!';
    Screen.Cursor:= crDefault;
    Stop;
    clstBox.Items.Add('Time consuming: ' +GetValueStr +' of: '+
          inttoStr(getFileSize(selectFile))+' File Size');
    Free;
end;
```

✌To understand threads one must think of several programs running at once. Imagine further that all these programs have access to the same set of global variables and function calls.

Each of these programs would represent a thread of execution and is thus called a thread. The important differentiation is that each thread does not have to wait for any other thread to proceed. If they have to wait we must use a synchronize mechanism.

All the threads proceed simultaneously. To use a metaphor, they are like runners in a race, no runner waits for another runner. They all proceed at their own rate.

☞Because Synchronize uses a form message loop, it does not work in console applications. For console applications use other mechanisms, such as a mutex (see graph below) or critical sections, to protect access to RTL or VCL objects.



2: 3 Threads at work with Log

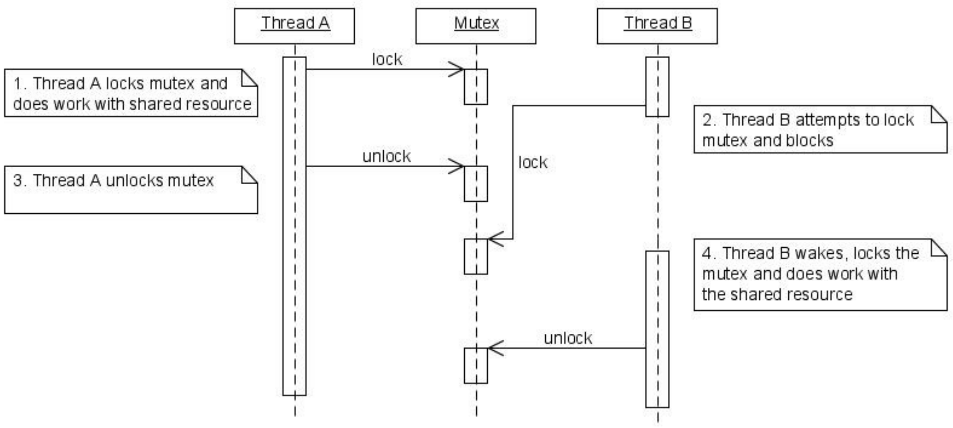
The point is you can combine asynchronous calls with threads! For example asynchronous data fetches or command execution does not block a current thread of execution.
But then your function or object has to be thread safe. So what's thread-safe. Because multiple clients can access for example your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables.

Tasks for advanced studies:

How can you crack a password with a massive parallel concept? Study all about a salt. A 512-bit salt is used by password derived keys, which means there are `2^512` keys for each password. So a same password doesn't generate always a same key. This significantly decreases vulnerability to 'off-line' dictionary' attacks (pre-computing all keys for a dictionary is very difficult when a salt is used). The derived key is returned as a hex or base64 encoded string. The salt must be a string that uses the same encoding.

We also use a lot of more multi scripts to teach and the wish to enhance it with a thread, simply take a look in the meantime at `141_thread.txt`, `210_public_private_cryptosystem.txt` and `138_sorting_swap_search2.txt`. At least let's say a few words about massive threads and parallel programming and what functions they perform.

Do not create too many threads in your apps. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.
You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.


Thread A
Mutex
Thread B
lock
1. Thread A locks mutex and does work with shared resource
2. Thread B attempts to lock mutex and blocks
unlock
lock
3. Thread A unlocks mutex
4. Thread B wakes, locks the mutex and does work with the shared resource
unlock

4: A real Thread with a synchronisation object

☞One note about async execution with fork on Linux with libc-commands; there will be better solutions (execute and wait and so on) and we still work on it, so I'm curious about comments, therefore my aim is to publish improvements in a basic framework on sourceforge.net depends on your feedback ;)

Try to change the sound file in order to compare two sounds at the same time:

```
04 pchar(ExePath+'examples\maxbox.wav'));
```

Try to find out more about the synchronisation object schema above and the question if it works in a synchronous or asynchronous mode.

⌨👍Check the source of LockBox to find out how a thread is used:

```
E:\maxbox\maxbox3\source\TPLockBoxrun\ciphers\uTPLb_AES.pas;
```

⌨☞Find another example to multiprocessing, the `QueensSolutions` is just one example; it shows the recursive solution to the 8 queens chess problem.

♏♒📪⍓❍ This is the last starter of the sequel 1-42. The work has been finished!

max@kleiner.com

Links of maXbox and Asynchronous Threads:

`059_timerobject_starter2_ibz2_async.txt`

`https://github.com/maxkleiner/maXbox3/releases`

http://sourceforge.net/projects/delphiwebstart
http://www.softwareschule.ch/maxbox.htm
http://sourceforge.net/projects/maxbox
http://sourceforge.net/apps/mediawiki/maxbox/

My Own Experiences:
http://www.softwareschule.ch/download/armasuisse_components.pdf

| | |
|---|---|
| SHA1: | maXbox3.exe F0AB7D054111F5CE46BA122D6280397A841C6FAB |
| CRC32: | maXbox3.exe 602A885C |