Debugging memory usage with kbmMW

The later versions of kbmMW contains more and more nice to have features for general logging, auditing, runtime exception handling with stacktraces and now also memory usage debugging.

These features are actually available for any application, even ones not using other parts of kbmMW.

Ive already been writing some articles about the logging and auditing system in kbmMW which also covered exception handling with stacktraces, but latest addition is the ability to realtime trace each and every memory allocation done by your application.

Why use kbmMW's memory debugger, when for example FastMM have leak detection build in?

FastMM only tracks memory allocations done via the regular GetMem etc memory allocations. It does not track allocations made via any of the Virtual/Heap/Global/Local allocation methods available in Windows.

Further kbmMW's debugger still works even if FastMMs leak detection is disabled, and provide features for logging memory use and allocations at any time in your application.

Starting out

You will need to add kbmMWDebugMemory to the uses clause of your application, and you will have to make sure that the following defines are set in your kbmMWConfig.inc file:

```
{$DEFINE KBMMW_SUPPORT_DEBUGMEMORY}
{$DEFINE KBMMW_INSTALL_DEBUGMEMORY_HANDLERS}
```

Otherwise all memory debugging will be disabled.

If KBMMW_SUPPORT_DEBUGMEMORY is omitted, then no memory debugging functionality (including all functions/methods) are available.

If KBMMW_INSTALL_DEBUGMEMORY_HANDLERS is omitted, then the memory debugging system will not automatically install the hooks and handlers, and thus the functions may be available (see above), but no memory tracing is happening.

When both defines are set, adding kbmMWDebugMemory to your uses clause of your application or unit, will automatically install kbmMW's memory debugging features and hooks.

To get the best out of it, you should also make sure your application is build with debug, stacktraces and an external TDS file alternatively a detailed MAP file. The produced *.tds or *.map file must be in the same directory as your executable when you run it if you want to do memory debugging outside the IDE.

Concept

kbmMW automatically hooks Borland type memory allocations, Microsoft Windows Virtualxxx, Globalxxx, Localxxx and Heapxxx allocation methods. It plays nicely with any 3rdparty memory manager including FastMM.

Each allocation and reallocation made, is assigned a unique incrementing 64 bit number by kbmMW. This number can be used for tracking all allocations made between two points in time, called checkpoints.

Basically a checkpoint is just an 64 bit number. This way you can check exactly what allocations have happened in subsets of your code, and even get a stacktrace to where the allocations took place.

Statistics

kbmMW maintains statistics over allocated memory and allocation counts.

These can be obtained at any time like this:

```
lLiveAllocationsCount.Caption:=inttostr(TkbmMWDebugMemory.LiveAllocationCount)+
      ' ('+inttostr(TkbmMWDebugMemory.LiveAllocationCountPerSec)+'/sec)';
lLiveAllocSize.Caption:=inttostr(TkbmMWDebugMemory.LiveAllocationSize)+
      ' ('+inttostr(TkbmMWDebugMemory.LiveAllocationSizePerSec)+'/sec)';
lMaxAllocationCount.Caption:=inttostr(TkbmMWDebugMemory.MaxAllocationCount);
lMaxAllocationSize.Caption:=inttostr(TkbmMWDebugMemory.MaxAllocationSize);
lMaxCapacity.Caption:=inttostr(TkbmMWDebugMemory.CurrentAllocationCountCapacity);
```

LiveAllocationCount is the number of active, in use, memory allocations detected. It counts allocations of all types (Borland - object/string/memory, Local memory, Global memory, Virtual memory, Heap memory). Since for example FastMM will allocate large chunks of memory via typically VirtualAlloc, and hand out pieces of this memory to applications using GetMem (Borlands memory manager interface), you will see an inprecise count (and size), since the count will include both the VirtualAlloc made by FastMM, and the individual GetMem (etc) calls made by the application.

Thus the live values are comparable values, but not exact values. You can depend on them to show for example increasing use of memory (perhaps indicating a leak), but you cant depend directly on the exact absolute value, as some allocations are counted twice due to the above situation.

Detecting leaks at shutdown

What is a leak? It's a resource that has been allocated, but which is never deallocated.

Some leaks are bad, some are not. The ones that are not bad, are leaks that are happening due to single, non repetitive allocations, typically made during startup of an application, which are never explicitly freed. In fact the RTL and VCL contains numerous such leaks.

Those leaks are not bad, because the operating system (Windows in this case) will automatically release all memory used by an application the moment the application shuts down.

The bad leaks are those that repeatedly allocates more memory, but never deallocates it again. These leaks will eventually make the application run out of memory space and/or make the system go extremely slow due to exhaust of physical RAM which results in paging. Paging is where currently less important memory segments are written to disk, to make room for currently more important memory segments, that are currently being allocated, or being read in from disk (page file).

It is normal for some paging happening on a system, but it is not normal if an application allocates as much memory as to slow all other processes significantly down, just due to paging happening.

A bad leak has occurred (repeatedly).

The only time to reasonably reliable to detect if those bad leaks has occurred, is at application shutdown time. Why? Because at that time you know that all (most) allocated memory should have been released by your applications destructors / FormClose events etc.

kbmMW makes this simple to check for. Some place, very early in your application's startup cycle, put these lines:

```
TkbmMWDebugMemory.ReportDestination('c:\temp\leaks.txt');
TkbmMWDebugMemory.ReportLeaksOnShutdown:=true;
TkbmMWDebugMemory.StartLeakChecking;
```

Now you have defined where leak reports should be dumped, that you want an automatic leak report to be created on shutdown, and that you want leak detection to start immediately. In fact all that StartLeakChecking does, is to load any TDS/MAP debug information (for stack trace purpose) and then register a baseline checkpoint from which it will check that allocations has been freed. All allocations before this time will be ignored and thus all build in VCL/RTL leaks and all objects allocated for the TDS/MAP info.

You can actually see the baseline value by checking:

```
ShowMessage('Baseline='+inttostr(TkbmMWDebugMemory.Baseline));
```

If you want the leak detection to include everything since the memory allocation hooks was installed. Set baseline to zero. Eg.

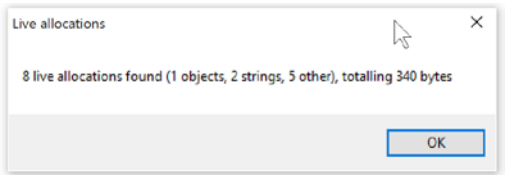
```
TkbmMWDebugMemory.Baseline:=0;
```

The checkpoint for last allocation made, can be obtained via:

```
var
  cp:TkbmMWDebugMemoryAllocationKey;
…
  cp:=TkbmMWDebugMemory.Checkpoint;
  ShowMessage('Checkpoint='+inttostr(key));
```

Now when you run your application and then closes it again, kbmMW will produce a report of non freed allocations.

Default it will present an overview status on screen like this:



And output details to the designated file:

```
Class:TkbmMWInnerThread Count:1 TotalSize:84
278956) Object:<unnamed> Class:TkbmMWInnerThread Addr:02F4C498 Size:84 hModule=00400000
<406A8A> : System(System.pas) : Line 4570 : @GetMem$qqri
<40791B> : System(System.pas) : Line 15975 : TObject.NewInstance$qqrv
<408126> : System(System.pas) : Line 17293 : @ClassCreate$qqrpvzc
<6085F6> : kbmMWGlobal(kbmMWGlobal.pas) : Line 8833 : TkbmMWCustomThreadPool.GetIdleCount$qq
<60899E> : kbmMWGlobal(kbmMWGlobal.pas) : Line 9199 : TkbmMWLockFreeHashArray32.Increment$qq
<608E71> : kbmMWGlobal(kbmMWGlobal.pas) : Line 9346 : TkbmMWLockFreeHashArray64.Increment$qq
<68E938> : kbmMWDebugMemory(kbmMWGlobal.pas) : Line 9176 : %TkbmMWLockFreeHashArray__1$44Kbm
<68E972> : kbmMWDebugMemory(kbmMWGlobal.pas) : Line 9179 : %TkbmMWLockFreeHashArray__1$44Kbm

278958) Unicode string:TkbmMWScheduledRelaxedEventThread(kbmMWSystemScheduler) RefCount:2 Ad
<4097D3> : System(System.pas) : Line 24231 : @NewUnicodeString$qqri
<40A71A> : System(System.pas) : Line 29853 : @UStrCatN$qqrv
<68C4BD> : kbmMWDebugMemory(kbmMWDebugMemory.pas) : Line 1243 : TkbmMWDebugMemory.Scan$qqrxp
<608736> : kbmMWGlobal(kbmMWGlobal.pas) : Line 8859 : TkbmMWCustomThreadPool.Reserve$qqrxo
<4BB92C> : System.Classes(System.Classes.pas) : Line 12585 :
<4BB98C> : System.Classes(System.Classes.pas) : Line 12585 :
<40971E> : System(System.pas) : Line 24006 : ThreadWrapper$qqspv
Address <0> unknown

279038) Unicode string:I RefCount:1 Addr:06AD0230 Size:16 hModule=00400000
<409C60> : System(System.pas) : Line 25201 : @UStrAsg$qqrr20System.UnicodeStringx20System.Un
<5F7549> : kbmMWDateTime(kbmMWDateTime.pas) : Line 2820 : TkbmMWDateTime.TrySetRFC1123DateTi
<5F9027> : kbmMWDateTime(kbmMWDateTime.pas) : Line 3654 : TkbmMWDateTime.$o20System.UnicodeS
<68D143> : kbmMWDebugMemory(kbmMWDebugMemory.pas) : Line 1495 : kbmMWDebugMemoryGlobalFree$c
<68D198> : kbmMWDebugMemory(kbmMWDebugMemory.pas) : Line 1496 : kbmMWDebugMemoryGlobalFree$q
<68D306> : kbmMWDebugMemory(kbmMWDebugMemory.pas) : Line 1520 : kbmMWDebugMemoryGlobalReAllo
<608D7D> : kbmMWGlobal(kbmMWGlobal.pas) : Line 9299 : TkbmMWLockFreeHashArray32.Decrement$qq
<68C5A6> : kbmMWDebugMemory(kbmMWDebugMemory.pas) : Line 1270 : TkbmMWDebugMemory.Scan$qqrxp

278927) Data(BORLAND) Addr:02F36A38 Size:64 hModule=00400000
<408770> : System(System.pas) : Line 17807 : TMonitor.GetMonitor$qqrxp14System.TObject
<4BB721> : System.Classes(System.Classes.pas) : Line 12585 :
<5CC939> : Vcl.Forms(Vcl.Forms.pas) : Line 10181 : Forms.TApplication.SetTitle$qqrx20System.
<5CC9BB> : Vcl.Forms(Vcl.Forms.pas) : Line 10204 : Forms.TApplication.SetHandle$qqrp6HWND__
<5CC9E9> : Vcl.Forms(Vcl.Forms.pas) : Line 10216 : Forms.TApplication.IsDlgMsg$qqrr6tagMSG
<5CBBF4> : Vcl.Forms(Vcl.Forms.pas) : Line 9768 : Forms.TApplication.CheckIniChange$qqrr24Wi
<5CBF1A> : Vcl.Forms(Vcl.Forms.pas) : Line 9833 : Forms.TApplication.WndProc$qqrr24Winapi.Me
<5CBF5D> : Vcl.Forms(Vcl.Forms.pas) : Line 9842 : Forms.TApplication.WndProc$qqrr24Winapi.Me

278835) Data(OS_LOCAL) Addr:01334398 Size:8 hModule=00400000
<53302E> : Vcl.Controls(Vcl.Controls.pas) : Line 15003 : Controls.TDockTree.PaintSite$qqrp5H
<52AD56> : Vcl.Controls(Vcl.Controls.pas) : Line 9822 : Controls.TWinControl.ControlAtPos$qq
<52AD6C> : Vcl.Controls(Vcl.Controls.pas) : Line 9825 : Controls.TWinControl.ControlAtPos$qq
<52AB31> : Vcl.Controls(Vcl.Controls.pas) : Line 9761 : Controls.TWinControl.SetParentWindow
<524F04> : Vcl.Controls(Vcl.Controls.pas) : Line 6004 : Controls.TControl.SetName$qqrx20Syst
<529A46> : Vcl.Controls(Vcl.Controls.pas) : Line 9036 : Controls.TWinControl.FlipChildren$qq
<52CA0C> : Vcl.Controls(Vcl.Controls.pas) : Line 10838 : Controls.TWinControl.DockDrop$qqrp2
<5C1B70> : Vcl.Forms(Vcl.Forms.pas) : Line 3247 : Forms.TScrollingWinControl.IsTouchProperty

278955) Data(BORLAND) Addr:02F21110 Size:12 hModule=00400000
<4404DB> : System.Generics.Collections(System.Generics.Collections.pas) : Line 2632 : Generi
<43F454> : System.Generics.Collections(System.Generics.Collections.pas) : Line 1815 : Generi
<43F479> : System.Generics.Collections(System.Generics.Collections.pas) : Line 1826 : Generi
<43E3FF> : System.Generics.Defaults(System.Generics.Defaults.pas) : Line 1513 :
<609718> : kbmMWGlobal(kbmMWGlobal.pas) : Line 9629 : TkbmMWLock.BeginWrite$qqrv
<6097C8> : kbmMWGlobal(kbmMWGlobal.pas) : Line 9664 : TkbmMWLock.BeginWrite$qqrv
```

As you can see, the debugger is able to determine if its objects, strings or other types of memory that has been leaked, and gives you in the first section, statistics over how many instances of a particular class are leaked. In this case just one instance of a TkbmMWInnerThread. This one is a safe leak, that just exists because the kbmMW scheduler have a relaxed event thread running handling events. The memory debugger has registered a scheduled event for calculating allocations/sec.

The stacktrace may be more or less precise, depending on the amount of debug information compiled into your application (stackframes must be enabled), and depending on if you let Delphi generate an external TDS or MAP file which kbmMW's stack trace functionality can use.

You can choose not to collect stacktraces for allocations. This will save some memory and CPU time, and will make your report less verbose. This can be done by adding the following line before the StartLeakChecking call:

```
TkbmMWDebugMemory.CollectStacks:=false;
```

Tracing allocations in specific situations

You may want to report all allocations made within a specific time interval, or between two locations in your code. You do this by using the checkpoint method to obtain a number at the two locations and then select what you want in your report. Eg.

```
FMyCP1,FMyCP2: TkbmMWDebugMemoryAllocationKey;
…
FMyCP1:=TkbmMWDebugMemory.Checkpoint;
<your interesting code>
FMYCP2:= TkbmMWDebugMemory.Checkpoint;
MyReport(FMyCP1,FMyCP2);
…

procedure MyReport(ACP1,ACP2: TkbmMWDebugMemoryAllocationKey);
var
   sr:TkbmMWDebugMemoryScanResult;
begin
     sr:=TkbmMWDebugMemoryScanResult.Create;
     try
        TkbmMWDebugMemory.Scan(sr,FMyCP1,FMyCP2,[mwdmstObject]);
        sr.Log(mwltDebug,mwllDetailed,[mwsrpoNoStack]);
     finally
        sr.Free;
     end;
end;
```

The above example will only report object instance allocations, and will report them via the kbmMW log system as debug log, without any stacktrace.

This is a small introduction into kbmMW's memory debugger.

Best regards

Kim Madsen