

# Re: Memory leak with CAsyncSocket::Create

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.mfc/2007-07/msg00747.html>

---

- *From:* Joseph M. Newcomer <[newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)>
  - *Date:* Tue, 10 Jul 2007 13:48:46 -0400
- 

That's a quite different problem; read my essay on how storage allocators work.

Mem Usage is relatively useless most of the time, because it is not actually a metric of how much memory is being *\*used\**, only about how much memory is *\*allocated\** to the process (which is quite different from the amount of memory that is actually in use at any given instant!). Yes, if it monotonically increases, you have either a leak or serious fragmentation issues, but that's about the only value it has: the first derivative, not the actual number, is all that matters. If you are seeing an increasing footprint without actual leakage, then you have a fragmentation problem, and that is a completely different problem than what you appeared to be describing.

See below...

On Tue, 10 Jul 2007 11:32:17 -0400, r norman <[r\\_s\\_norman@xxxxxxxxxxxxx](mailto:r_s_norman@xxxxxxxxxxxxx)> wrote:

On Tue, 10 Jul 2007 09:45:46 -0500, "AliR \(\VC++ MVP\)" <[AliR@xxxxxxxxxxxxx](mailto:AliR@xxxxxxxxxxxxx)> wrote:

Thank you for your help. Perhaps "leak" is the wrong word because my programs in debug mode never show a problem on exit. However the Create method is consuming system memory that is not released back to the system until the program closes.

\*\*\*\*\*

Yes. That is what is supposed to happen. If memory is allocated to your PROCESS then it is nominally allocated forever. The REAL question is why there appears to be a need to keep allocating more memory to your process when you are freeing storage up. That is actually a very important question to discover the answer to.

\*\*\*\*\*

The memory consumption is either shown as "Mem Usage" on the Task Manager Processes page or by the following code:

```
PROCESS_MEMORY_COUNTERS pmc;
HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION
| PROCESS_VM_READ, FALSE, _getpid());
size_t retval = 0;
if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))
```

## Re: Memory leak with CAsyncSocket::Create

```
retval = pmc.WorkingSetSize;  
CloseHandle(hProcess);
```

The memory used by my process continues to grow and grow indefinitely even though I close and delete the socket. That seems intolerable, whether or not it officially counts as a leak.

Maybe the Windows memory manager will eventually discover this and recoup the memory, but how long will it take?

\*\*\*\*

Forever. There is "Windows Memory Manager" that will make such a discovery. Not in application space, and not in the kernel. There are many, many levels of allocation going on here and it is essential to distinguish them. There is the allocation of physical memory, which is handled by the only component I would refer to as the "Windows Memory Manager". Physical memory is allocated on demand and pages which have not been used in a long time (that is, are not part of the working set) may be removed from physical memory and left instead on the paging file or in the executable image so that other pages which are more in demand may be moved in. This deals exclusively with how the operating system manages the physical memory of the computer, and has nothing to do with how applications manage their own memory.

Application memory is allocated by VirtualAlloc and freed by VirtualFree. There is no other way to get memory allocated to your application. However, these allocations are fairly coarse-grained (64K on a typical installation) and would be unsuitable for ordinary allocation problems. Therefore, the heap allocators manage this memory by allowing it to be split into smaller pieces (HeapAlloc). Various strategies such as modified-quickfit and such are used to try to minimize fragmentation, but they cannot prevent fragmentation. They merely reduce its effects somewhat. But there is no mechanism for returning this heap memory to the operating system in the default heap (and it is not clear what HeapDestroy on custom heaps actually does, but I suspect it is the only situation in which it *would* be possible to return memory to the operating system). So what you are left with is the issue of how memory is allocated internally in your app. You can use LocalAlloc/LocalFree, although those are not particularly good choices; you typically use malloc/free, or in C++ these are typically wrapped in new/delete. These functions manipulate the blocks of storage allocated to your app via VirtualAlloc, breaking them into small pieces and handing those pieces out and eventually returning them. There are various strategies here: quickfit, modified quickfit, first-fit, best-fit, split-reduction, and combined with strategies such as aggressive-coalescing and lazy-coalescing, these give different performance and fragmentation profiles.

Generally, tuning performance for memory-intensive applications takes about as long as it took to develop the application in the first place; I've literally spent months taking code that came to me and making it perform well by reducing the costs of memory allocation, and most commonly reducing the effects of memory fragmentation. In some cases, major rewrites were required because the performance issues had nothing to do with memory footprint at all, but on working set size, and paging was killing them. It isn't a simple task; you can spend several weeks just gathering data and analyzing it. What is your percentage of fragmentation? (Ratio of unused small blocks to total allocation). What is the size distribution of small blocks? What is the size distribution of all blocks? What is the memory span? (This refers to how many pages you would have to touch

## Re: Memory leak with CAsyncSocket::Create

to process data in one object, and/or the number of pages you would have to touch to iterate across a set of objects). When does it make sense to preallocate buffers inline (TCHAR something[FIXED\_SIZE]) vs. having pointers to buffers (LPTSTR, CString, std::string)?

\_heapwalk will become overly familiar to you. I tend to write files that can be imported to Excel and used to graph what I need to look at. Or which can be processed by some scripting language into some new set of data to be shown in Excel. You can spend a couple weeks gathering data, just so you can spend a couple weeks staring at it, just so you can start to form insights in what you need to \*actually\* gather to do the job, then gathering \*that\* data, then staring at it, and finally seeing where the memory is going. Pre-MFC/C++ I actually wrote my own allocation interfaces that took \_\_FILE\_\_ and \_\_LINE\_\_ and a memory tag, and tagged each block of memory (you get this free in the debug allocator now!) so I knew at any given point exactly how much memory was in use, and why. I would do regular snapshots and watch the balance change over the execution lifetime, and say "OK, how can I change this...". Back in the days of MS-DOS, I wrote out data which I processed with a scripting language called "PostScript" and it plotted memory usage on my laser printer (I have been an expert PostScript coder in the past), and the pictures were quite interesting. I am tempted at times to see if I can recover that code from old backup tapes and apply it to my new color laser printer to get REALLY interesting shots, although these days I could probably write the algorithm faster in MFC. Or use Origin (a graphics package) to do 3-D plots of data allocation over time (it produces VERY sexy graphs).

None of this is really straightforward, all of it requires a fair amount of understanding of the application and its storage usage patterns, and it all requires significant effort to instrument and study (I haven't looked at the performance analysis tools of VS2005 yet, some of this may already be there, but gathering numbers and looking at numbers are separate tasks, and deciding the best strategy to act upon the numbers is yet another phase). None of this relates to the "Windows Storage Allocator" or HeapAlloc, malloc, or new directly; they may be sound, robust, thread-safe, etc., and STILL fragment you like crazy if you hit them with the wrong patterns.

This is one of the reasons I rarely worry about optimizing code these days; real applications need data optimizations, and they are far harder.

Note that a third-party allocator is still not going to solve the basic problem, although by using variants of quickfit and tuning it for either lazy or aggressive coalescing may give you a result more tuned to your application's needs, so I wouldn't rule it out. But it won't change the fact that in C/C++, there can be no compaction, nor can there exist any mechanism to "return storage to the operating system" that could be really effective. In general, any mechanism that could be invented to do this is likely to actually reduce performance by a larger factor than it contributes to improving performance (since the smallest unit that can be freed is 64K, it would require a contiguous 64K-aligned chunk have no objects in it, a situation unlikely to arise in common practice).

There are no easy answers. They all take time. If you want to do such optimizations, budget yourself a couple months to do them. Not counting the time taken to retest everything after you've rewritten the algorithms to improve the performance.

joe  
\*\*\*\*\*

## Re: Memory leak with CAsyncSocket::Create

All I know is that I have a problem in the field when my program runs month after month and this is the best clue I can generate on my test bed about what might be the cause. I now have a version running in the field that logs its memory usage once a day, but that will take days and weeks to analyze. The problem is that the CAsyncSocket delete and recreate only occurs when there are problems in the field, and they occur only sporadically.

Incidentally, the memory consumption occurs both in debug and release compilations and for both Unicode and simple char string (Character Set not set) options. I tested that because CAsyncSocket::Create uses T2A\_EX within CAsyncSocket::Bind and I wonder whether that might be the problem.

I couldn't find a memory leak. What you are most likely seeing is windows memory management doing its work. The Create method is creating a socket object, when when it's freed the memory is not given back to the system right away.

(I used this method to detect a leak)

```
// Declare the variables needed
#ifdef _DEBUG
CMemoryState oldMemState, newMemState, diffMemState;
oldMemState.Checkpoint();
#endif

for (int i=0; i<10; ++i)
{
CAsyncSocket *pAS = new CAsyncSocket;
pAS->Create();
pAS->Close();
delete pAS;
}

#ifdef _DEBUG
newMemState.Checkpoint();
if( diffMemState.Difference( oldMemState, newMemState ) )
{
TRACE( "Memory leaked!\n" );
}
#endif

AliR.
```

"r norman" <r\_s\_norman@xxxxxxxxxxxx> wrote in message

Re: Memory leak with CAsyncSocket::Create

news:2895939efidggi556s7fbje0euhm2jd2d0@xxxxxxxxxxx

I have traced a memory leak problem to CAsyncSocket::Create(). Is this a known problem? Is there a workaround/solution/fix? Here is sample code:

```
for (int i=0; i<m_nReopenCount; ++i) {
CAsyncSocket *pAS = new CAsyncSocket;
pAS->Create();
pAS->Close();
delete pAS;
}
```

Running this 1000 times uses up 1200 KBytes of memory, or just over 1 KByte per call. Commenting out the Create() leaves memory clean. (And please don't complain about my bracketing style -- I like it.)

I have Visual Studio 2005 Professional version 8.0.

Incidentally, I also discovered that the call to Create() is not re-entrant. My application involves connecting to some 10 to 20 external devices and my normal code creates a CWinThread to support each socket, where the socket is created and destroyed only within the thread routine. Creating all the threads and starting them up simultaneously meant having multiple instances of CAsyncSocket::Create() being called at the same time, crashing my system (memory access faults). That one I found and fixed with sentries. Now I am left with the memory leak.

The problem is that I have an rather intricate communication protocol system all self contained so that adding a new hardware device simply means creating a new instance of the whole works. It runs fine until the external hardware goes haywire, in which case I destruct the whole instance and start a new one which breaks and reconnects the socket with a clean start and, most of the time, results in a good connection; the external device resets itself through the

Re: Memory leak with CAsyncSocket::Create

disconnect.

One faulty device, though, generated thousand upon thousand of disconnects over a number of days and, after a few hundred thousand of these I my own system crashed due, I have now found out, to a lack of memory caused by this leak.

My application must run essentially as an embedded system, unattended week after week, month after month so I cannot tolerate a memory leak.

Does anybody know about this? Is there a simple clean way to force a socket disconnection on a CAsyncSocket and then reconnect? My application is the connect() end of the socket, not the listen() end.

Joseph M. Newcomer [MVP]

email: [newcomer@xxxxxxxxxxxxx](mailto:newcomer@xxxxxxxxxxxxx)

Web: <http://www.flounder.com>

MVP Tips: [http://www.flounder.com/mvp\\_tips.htm](http://www.flounder.com/mvp_tips.htm)

.