Upgrading Computers for Memory-Hungry Applications

Binglong Xie
binglong.xie@siemens.com
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540

Abstract

Having enough physical computer memory for a data intensive application is crucial for its performance to avoid page swapping to and from the disk. The first approach is to add more memory to a current 32-bit x86 PC. This approach gives the application a maximum of 3 GB user memory address space, and generally the application can only make use of that amount of physical memory at most. If more memory is needed, upgrading to an x64 PC is recommended. Theoretically there is almost no limitation to the user memory address space with x64, and the physical memory available to the application is much more a direct effect of adding physical memory without the address space barrier.

1. Introduction

Having enough memory for a data intensive application is crucial for its performance. When not all the data/code can be held in physical memory, a modern virtual memory operating system automatically uses disk paging files in the background to simulate the physical memory. Because the disk is much slower than physical memory, page swapping slows the application significantly. It is therefore very important that the application can get enough physical memory.

To see how page swapping can hurt an application, let's have a look at the speeds of the RAM (Random Access Memory, i.e. physical memory) and the hard drive respectively. There are a lot of different RAM types, and what is widely used in the PC world is a category called DRAM (Dynamic RAM), which is relatively easy to get a high capacity but needs refreshing from time to time to prevent data loss (thus called dynamic). The DRAM varies from out-of-date FPM-DRAM (Fast Page Mode DRAM), EDO-DRAM (Extended Data Out DRAM), SDRAM (Synchronous DRAM), to relatively recent RDRAM (Rambus DRAM), DDR-SDRAM (Double Data Rate SDRAM) and DDR2-SDRAM. For a typical 667MHz DDR2-SDRAM module, the data transfer speed is rated at PC2-5300, i.e. 5300MB/s (667MHz, 8-byte width). However, for a typical SATA/PATA (Serial/Parallel Advanced Technology Attachment) hard drive, the data transfer rate varies between 20MB/s and 60MB/s, depending on whether it is a read or write operation. The hard drive is about 100 times slower than the RAM. Even worse, the hard drive has an access time, about 10ms, to move its magnetic head to the right location if it is not there yet, before it can start transferring data. On the other hand, the 667MHz DDR2-SDRAM only needs a few cycles to start data transfer, where a cycle is about 3ns, or 0.003ms. Even though there are different caches for both the RAM and the hard drive,

the overall conclusion is that extensive page swapping with a hard drive file should be avoided by all means.

What kinds of applications are memory-hungry? Well, any application that handles large amount of data is a good candidate of memory hunger. This includes a lot of server applications, such as Microsoft Exchange Server, Microsoft SQL Server, etc. But do not assume that memory hogs are far from you when you are working on a workstation. No matter if you are a statistician, chemist, or a computer vision researcher, you may find your application is well within this category. For example, in computer vision, machine learning is often used to train a classifier from a large number of training samples (e.g., tens of thousands), and on each training example there are thousands of features used. Such a training job needs a lot of memory, and without enough physical memory the training can last for days. It is often forced to use less training samples and features for faster training, at the cost of a less effective classifier as a result.

It is clear that adding more physical memory to the computer is necessary for the memory-hungry applications.

2. Overview of 32-bit PC Memory Hierarchy

A lot of current PC systems are based on 32-bit processors, such as Intel Pentium 4 and AMD Athlon. They are 32-bit processors, because the data width (i.e., the native operand width) is 32-bit. These PCs run Microsoft Windows 2000/XP/Vista 32-bit version. Our application runs on top of the 32-bit Windows OS (Operating System).

Isn't it obvious that we add enough physical memory to the computer and we are done? Is it true that, if I add 8GB of RAM to the computer my application can then fully use 8GB for its huge data? Actually it is not so simple. To really understand how the application gets the physical memory, we need to look step by step into the hierarchy of memory management of a 32-bit PC, in aligning to our purpose of answering the question.

2.1 Physical RAM

Firs of all, you need to check your mother board specifications, and find out how many RAM slots are available onboard, what is the maximum capacity that each slot can support, and the maximum physical RAM that a mother board can support. It is absolutely meaningless to add more memory than the mother board can support. In a lot of 32-bit systems, you cannot add more than 4GB of RAM. But some of them may allow you to add at most 64GB to it.

2.2 CPU Physical Address Space

A CPU (Central Processing Unit, or processor) accesses the RAM by giving address signals on its address pins. The physical address space consists of all possible combinations of the address pin signals. Earlier 32-bit CPUs have 32 address pins (2^32=4GB physical address space). Later Pentium series CPUs have 36 address lines (2^36=64GB physical address space), and new AMD CPUs have 44 address lines (2^44=16TB physical address space). The 32-bit CPUs normally only use 32-bit address

width and the extra pins are not used. However, if the PAE (Physical Address Extension) feature of the CPU is enabled, these CPUs can fully use all the address pins and access memory units beyond 4GB.

The mother board is designed taking into account the CPU's addressing capability, so it's not the case that you add RAM there but the CPU can only access part of it, with one important exception. The exception comes from memory-mapped devices and ROM in the system. The memory-mapped devices, such as a video adaptor in every PC, and POST ROM, also take certain portion of the physical address space, and can push some RAM out of the physical address space such that not all RAM is available to the CPU. For example, if the CPU has 4GB physical address space, and you add 4GB RAM to the system, you are bound to lose a few MB, maybe more than 100MB of RAM, because of the memory-mapped devices and ROM. Your system may report RAM size smaller than 4GB. In such a case, the accessible physical RAM is limited by the physical address space.

2.3 OS Physical Address Space and Physical RAM Available to OS

Even if a CPU may support 16TB physical address space with its PAE enabled, the operating system may not. In reality, 32-bit Windows NT[1] (including Windows 2000/XP/Vista in this context) normally supports 4GB physical address space. Only when 32-bit NT's PAE feature is enabled (which in turn enables the CPU's PAE), it supports more than 4GB. Windows NT limits its supported physical address space in 64GB. The OS physical address space is a subset of the CPU physical address space. Within the OS physical address space, you can deduct the portion that is used by memory-mapped devices and/or ROM, and the rest is the maximal physical RAM available to the OS. The physical RAM available to the OS is what our system is running on, and this number is the one you really want to maximize when you add RAM modules to your motherboard.

2.4 Virtual Memory

An application normally does not use physical memory directly. Mostly since the introduction of Intel 80386 CPU, the PC architecture has been redesigned that there is a new layer, called virtual memory, standing between the application (and actually most parts of the OS itself), and the physical memory.

[1] Windows NT is a family of fully 32-bit operating systems by Microsoft for business users, including Windows NT 3.1, Windows NT 3.5, Windows NT 3.51 and Windows NT 4.0. Later versions (including 64-bit versions) have not been marketed as Windows NT, but internally they are on the NT basis, including Windows 2000 (NT 5.0), Windows XP (NT 5.1), Windows Server 2003 (NT 5.2) and Windows Vista (NT 6.0). Windows NT is different from Microsoft's consumer-oriented 16-bit/32-bit hybrid operating systems, such as Windows 3.x and Windows 9x/Me, which have come to an end and merged to the NT tree, and have already been replaced by Windows XP Home and hopefully by newly released Windows Vista Home editions.

2.4.1 Virtual Memory Manager

The Virtual Memory Manager (VMM) is one of the most important parts of modern operation systems. The Windows NT VMM is a kernel component. It is responsible to create seemingly much larger total memory available to running processes than the physical memory. The VMM partitions the physical memory into *pages* (4KB each by default), and the pages are backed by the disk files. The VMM itself lives in the *nonpaged pool*, which is the physical memory never paged to disk, due to its critical role.

The application always accesses memory using a virtual address. The virtual address is translated using the page directory and page table to get the page without the awareness of the application. In the translation process, there are two possibilities:

> a). The page is in physical memory, so the translation is successful, and the application can access the memory location;
>
> b). The page is not in physical memory. A page fault happens, and the VMM takes the control. The VMM finds an available physical memory page, loads the page from the backing disk file, and resumes the application code to access the memory location.

Because the access to memory happens in sequel, the VMM is able to use a small amount of physical memory to service accesses to large amount of pages on disk, thus creating virtual memory that can be far bigger than physical memory. With a few exceptions, it is actually much like that the virtual memory is always in a big paging file, and the physical memory is just like cache of the virtual memory.

The VMM has an important data structure called the *page-frame database*. This database contains the status of each physical memory page. The NT VMM periodically steals used physical memory pages from processes to maintain a healthy level of free physical memory pages to satisfy future memory requests. It does this in two steps:

> a). First, it marks the page as *transitional* and monitor if page faults happen for that page. If that happens it just return the page back to the process at no cost.
>
> b). Second, if it sees that the process does not ask for it in long time, it concludes that it's safe to steal it. Then it flushes the page to paging file and marks it free and available to future page requests.

The total effect of VMM is that if there is insufficient physical memory for data or code, it kicks out the data and code that are not accessed recently to the paging file on disk, and the system still runs with reasonable performance.

2.4.2 Total Virtual Memory

The total virtual memory of the system can be defined as the space allocated to hold data or code to all the processes running at any instant. The total virtual memory is a dynamic amount while the system is running. Generally, it consists of three portions:

a) Paging files
In 32-bit Windows NT, there can be at most 16 paging files. The paging files are the common backing of virtual memory.

b) Memory mapped files
Memory mapped files are generalized paging files. Because the whole virtual memory mechanism, it is easy to use a general disk file instead of a paging file as backing. However, memory mapped files are not managed and accessed like the memory; it has to be used through a special API (Application Programming Interface) with a file mapping object. A file mapping object, however, can also be created on the paging files.

c) AWE (Address Windowing Extensions) memory
AWE is a method to allocate physical memory directly in an application. We will cover that later.

As you can see, the total virtual memory is not limited by physical address space or physical memory available. Rather, it is limited by mostly the disk space.

2.5 Virtual Address Space

In 32-bit Windows NT, each process has full flat 32-bit virtual address space, that is, 4GB. That means any pointer in the process cannot be larger than 32-bit. Windows NT distinguishes the user mode process and kernel mode process. In the user mode process, its address space is less than 4GB; normally it is the lower 2GB, with some exceptions, no larger than lower 3GB. The upper 2GB or 1GB is only used for the kernel. Because each process gets its own separate virtual address space, and Windows NT can create many processes, the total virtual address space to all processes is unbounded.

The virtual memory that a user process can access may be larger than its virtual address space. For example, the application can create a file mapping object that spans larger the 4GB virtual address space, either through a memory mapped file or on paging files, given that the disk space is enough or the paging file space is available. Notice the function `CreateFileMapping()` needs two DWORD arguments to represent the file mapping object's maximum size, i.e., the size can be larger than 4GB. The application can allocate physical memory through AWE so that the total memory for a process could be larger than 2GB/3GB virtual address space.

However, the virtual memory that a user process can access *simultaneously* is always limited by the user virtual address space size, 2GB/3GB. Even if the process can own more virtual memory, it has to slide the virtual memory into the window of the virtual address space before it can access it. That means extra effort than just dereferencing a pointer. For example, before you can access the virtual memory of file mapping, you have to call `MapViewOfFile()` to map it into the virtual address space; in the case of AWE, you have to call `MapUserPhysicalPages()` to map the AWE memory into the virtual address space. This results in non-portable and hard-to-maintain code.

2.6 Win32 Virtual Memory API

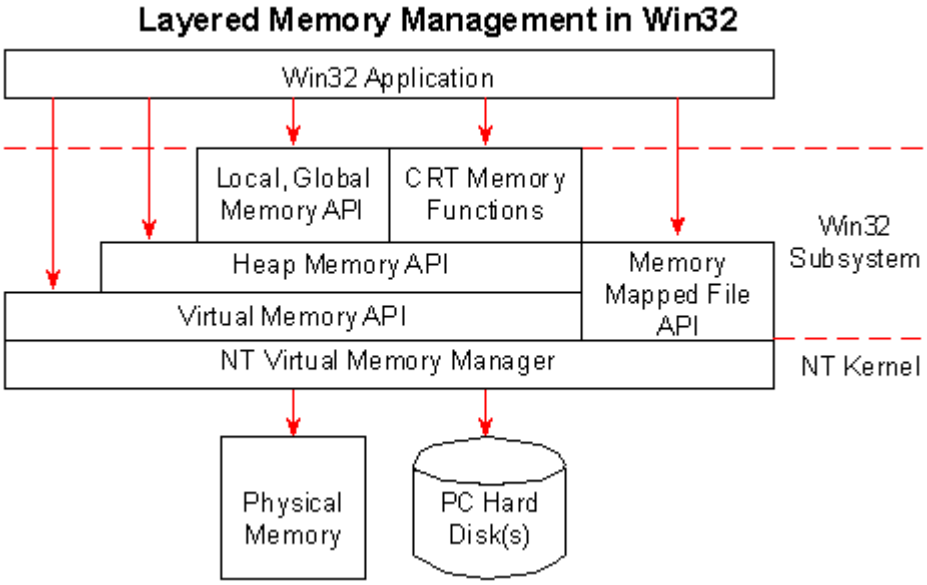

Figure 1: Layered Memory Management in Win32 (Image from MSDN)

The NT kernel actually supports other subsystems, but our interest is only on Win32 subsystem. From Figure 1, you can see that both Virtual Memory API (VMAPI) and Memory Mapped File API talk to NT VMM directly. Since we are more interested on the portable memory programming, we focus more on the VMAPI.

The VMAPI distinguishes two resources you can allocate: the virtual address and the virtual memory, both in pages (4KB by default). Both resources are managed through VMAPI functions such as `VirtualAlloc()`/`VirtualFree()`.

The virtual address that you can allocate with VMAPI is limited by the user virtual address space, and any other already allocated address blocks. Allocating the virtual address is also called *reserving* memory address, which is a cheap and fast operation. What it really does is to create an entry in the VAD (Virtual Address Database) of the process, marking that the address block is reserved. This step is useful if you want to have a continuous virtual address block before other memory requests in the same process get their hands on there.

You can also allocate virtual memory in pages with VMAPI, and this is called *committing* memory. The committed memory could be in either physical memory or the paging files. Be careful, committed memory does not mean physical memory. When you access a committed page, a page fault happens if it is currently only in the paging file. Figure 2 shows an example of various memory amounts, where Commit Charge Total is the total committed memory in the system.

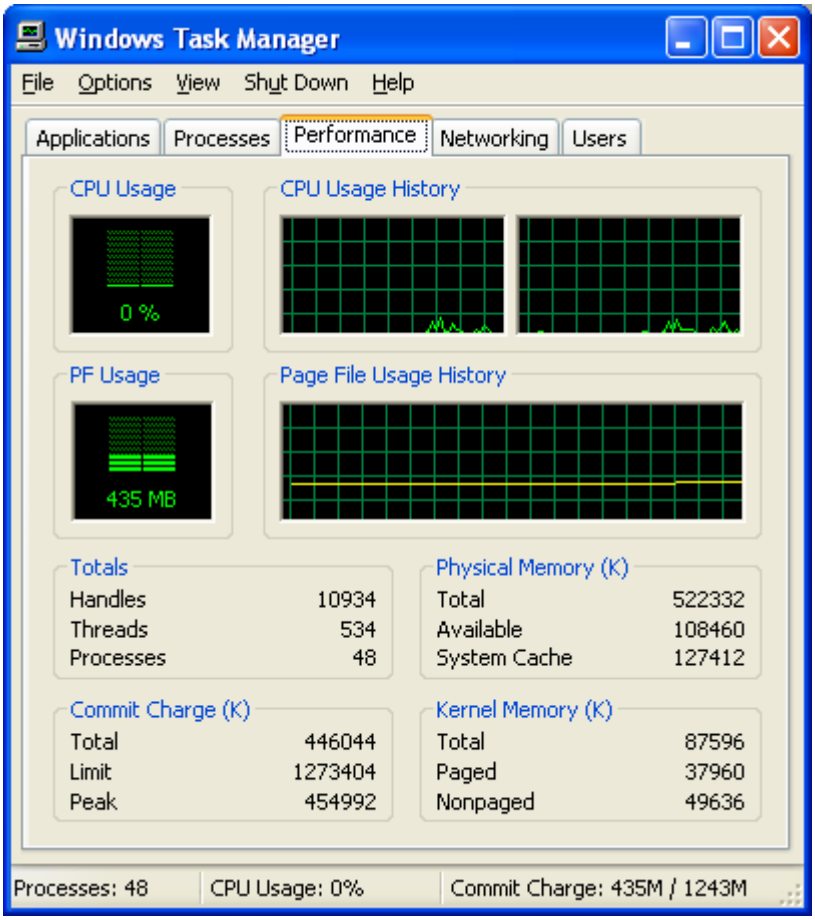

Figure 2: Example of Various Memory Amounts (Screenshot)

2.7 Address Windowing Extensions (AWE)

As the name suggests, AWE windows the physical memory into the virtual memory address space. AWE is the only way that a user mode application can allocate guaranteed physical memory, which is normally the privilege of kernel components, such as device drivers.

Just like VMAPI, AWE is also a low-level feature of Windows NT, and has to be used with VMAPI. The prerequisite of using AWE is that your application must have the **Lock Pages in Memory** privilege. To obtain this privilege, an administrator must add Lock Pages in Memory to the user's User Rights Assignments.

In your application, `AllocateUserPhysicalPages()` is used to allocate physical memory pages. The VMAPI function `VirtualAlloc()` with arguments `MEM_RESERVE|MEM_PHYSICAL` and `PAGE_READWRITE` is used to get a virtual address block. `MapUserPhysicalPages()` is then used to attach the physical memory to the virtual address block, and then the application can access the allocated memory.

Another benefit of AWE is that you can allocate a lot of physical memory, only limited by the physical memory available to the OS. That says, if the OS sees more than 4GB, the AWE can also possibly allocate more than 4GB, up to the Windows NT limit of 64GB, with NT's PAE enabled. However, the AWE physical memory accessible to your application at any instant is less than the user virtual address space of course.

2.8 Win32 Heap Management

Heaps provide memory management that isolates the OS dependent low-level details from an application. For example, a virtual memory system uses a page as the smallest memory allocation unit (in Win32, 4KB), but a heap would allow you to efficiently allocate much smaller objects, for example, an integer of 4 bytes. Win32 Heap Management is the direct layer over VMAPI and is the general heap for all other heap types on Win32.

A Win32 heap is used through `HeapAlloc()`/`HeapFree()`. Multiple Win32 heaps can be managed through `HeapCreate()` and `HeapDestroy()` in each process. Having multiple heaps provides a powerful means for special purposes. For example, you can use a separate heap for some functionality and do not track all allocations, and in the end you can just destroy the heap and all the allocations are automatically reclaimed. For each Win32 process, Windows creates a default heap, which can be obtained by calling `GetProcessHeap()`.You can instruct the linker the committed and reserved sizes of the default heap for faster/smoother heap growing, but they are not meant to limit the heap size. The heap can grow to the extent only limited by virtual address space or virtual memory size. Upon process termination, all Win32 heaps in the process are automatically destroyed.

2.9 Global and Local Memory Functions

For historical reasons, there are two sets of memory management functions, the global and local memory functions for managing the default heap of a Win32 process: `GlobalAlloc()`/`GlobalFree()`, and `LocalAlloc()`/`LocalFree()`. Both actually use `GetProcessHeap()` to get the default heap and then use Win32 heap functions `HeapAlloc()`/`HeapFree()` to access the heap, so they are actually identical in Win32. If your application does not use more than one heap, using either `GlobalAlloc()`/`LocalAlloc()` is enough.

2.10 The C Run-Time Heap

Your application is a program written in a programming language, or in multiple programming languages in mixture. Nevertheless, the native API of Win32 is in the C programming language. So I assume our application is written with C.

By default, the C memory functions `malloc()`/`free()` and C++ operators `new`/`delete`[2] give you access to the C Run-Time (CRT) heap. This is probably the

[2] C++ is open to usage of other memory instead of the CRT heap for the new operators implemented by the compiler. In addition, the user application can also overload the class-specific and/or global new operators to use other underlying memory management.

most used heap in portable applications. In Win32, the CRT heap is also implemented using the default heap of Win32 process. If you only use one heap in your application, there is no reason to not use `malloc()`/`free()` or `new`/`delete` for maximal portability. `malloc()` is equivalent to `GlobalAlloc()` with `GMEM_FIXED` flag so the return is an pointer instead of a handle that requires locking before accessing memory.

3. Maximize Memory for Your Application

Get back to the question, how can you get more physical memory for your Win32 application?

3.1 Comparison of Memory Allocations in an Application

Before we can answer the question, let's first look how an application obtains memory programmatically. As being examined in the previous section, an application has various approaches to get memory, as listed in Table 1.

From Table 1, it is clear that AWE is the only method that you can get guaranteed physical memory, which is capped by the available physical memory to the operating system. However, there are two problems with it. First, AWE is not portable, so your application can not be compiled for example on Linux. Second, because the virtual address space is between 2GB and 3GB, the application has to use mapping/remapping to access AWE physical memory larger than the virtual address space, and this is very tedious.

File mapping can get you virtual memory more than the virtual address space too, but again mapping/remapping is needed. And, it is also platform dependent.

Table 1: Different Methods for Application Process to Allocate Memory

Memory Allocation Methods		*Virtual address space*	*Memory maximum*	*Memory Type*	*Portability*
VM API		2GB/3GB	2GB/3GB	Virtual	Win32
Heap	Win32 heap	2GB/3GB	2GB/3GB	Virtual	Win32
	Global/Local heap	2GB/3GB	2GB/3GB	Virtual	Win32
	CRT heap	2GB/3GB	2GB/3GB	Virtual	ANSI / ISO
File mapping	On regular file	2GB/3GB	Virtual	Virtual	Win32
	On paging files	2GB/3GB	Virtual	Virtual	Win32
AWE		2GB/3GB	Physical	Physical	Win32

Among all the heap memory management methods, the obvious winner is the CRT heap if you do not need multiple heaps in a process, because it is the only portable one. With

the CRT heap a Win32 process can get at most 2GB/3GB virtual memory without mapping/remapping.

VMAPI is a low-level API, both coarse and non-portable for an application, and is not normally used.

As observed, it is common for all methods that the 2GB/3GB virtual address space limitation plagues an application. If the application uses less than so much memory, that is fine; otherwise, painstaking mapping/remapping has to be carried out to bring other virtual/physical memory into scope. This is unfortunate for Win32 applications.

If you can afford mapping/remapping, as well as non-portability, AWE may be your choice when you need a lot of physical memory. File mapping can also give you a lot of memory but it is virtual memory and could be on disk.

In other situations you will use the CRT heap, because it is portable and does not provide less functionalities than other heaps or VM API for a normal application. Our interest is actually mainly on the CRT heap memory.

3.2 3GB Virtual Address Space

As mentioned before, the virtual address space of a Win32 user mode process is normally 2GB, from 0x0000 0000 to 0x7FFF FFFF. There are some spots that are not available to the application process, but let us skip the details for now. The kernel uses the upper 2GB, i.e., 0x8000 0000 to 0xFFFF FFFF.

We talked about 3GB, but that is not automatically available. That is, in general you will not be able to allocate memory more than 2GB, nor will you get a pointer beyond 0x7FFF FFFF. If you have such a pointer and you try to access memory there, you get memory access violation and the process is terminated by Windows.

Since 2GB is not really a large number for memory hog applications, we really want to push the limit to 3GB. There is a special boot option, /3GB, for some versions of Windows NT. When this option is added, Windows boots with its kernel squeezed in upper 1GB virtual address space (0xC000 0000 to 0xFFFF FFFF), so that the lower 3GB (0x0000 0000 to 0xBFFF FFFF) is available to each application. Now the applications are given the opportunity to allocate more than 2GB, up to 3GB virtual address blocks. Warning: In some Windows versions, /3GB only compacts the kernel to 1GB address space, but does not really give 3GB address space to the user process. These versions are only used to test the kernel mode components such as device drivers.

For an application to be able to utilize the extra 1GB virtual address space, however, it must be linked with **/LARGEADDRESSAWARE** link option, otherwise it only sees the normal 2GB address space available. If what you get is an executable, you can use the tool ImageCfg or EditBin to set the corresponding flag in the executable file such that it is large address aware. In both cases, you need to make sure that the pointer arithmetic in the application is compatible with 2GB+ memory space, i.e., the executable code must

not assume that all pointers live below 2GB. Normally this is about overflow, signed/unsigned operations etc. related to the pointers. It is probably dangerous to just enable that flag and assume the application can use 3GB.

However, /3GB does not come for free. Using /3GB may give the Windows kernel a hard time. The kernel has to shrink its stuff into a smaller space. The biggest victim is the kernel mode file cache, and it is bounded to be shrunk if /3GB is used. So if your application uses a lot of file I/O, even if you get more memory space, you may lose the performance here. Sometimes, a video card driver needing a big kernel aperture (memory used to communicate between the video card and main memory), e.g., a few hundreds MB, may fail to load. In other instances, when the kernel tries to allocate some memory and it could not fit into the 1GB, you get the doomed blue screen error. These all say that you may experience an instable system with /3GB.

Windows provides another complementary boot option, /USERVA, to help. With /USERVA, you can fine-tune the virtual address space to the user process, between 2048MB (2GB) and 3072MB (3GB), so that the kernel can get more than 1GB to run stably. For example, /3GB /USERVA=2900 option gives user processes 2900MB address space, and kernel gets 1196MB, instead of 1024MB. You can experiment this number and stop at a point that both your OS and your application are happy hopefully.

One thing you may want to follow is, unload all unnecessary devices, close all unnecessary applications, and yield more memory and resource for both your application and the rest of system.

3.3 What Is Available in 3GB Virtual Address Space

Even given /3GB boot option, if you think you can really `malloc()` 3GB memory for your application data, you are wrong. Before your first line of code in `main()`, the entry point of your C/C++ application, runs, some spots of the user virtual address space has already been taken. Below is just an incomplete list:

a). There is a *no man's island* of 64KB near the 2GB boundary (slightly below 0x7FFF FFFF) and the virtual address is never allocated there (a burden coming from Windows NT common code base for also Alpha AXP processors).

b). Before your process starts, Windows must set up the environment variable block and other process related data (such as data structure for the default heap) in your virtual address space.

c). The executable of your process takes memory address space. Normally it is loaded at 0x0040 0000.

d). Your application always needs to call operating system services (that is what the OS is supposed to do). There are the Win32 subsystem DLLs (Dynamic-Link Library) interfacing the application to the kernel. They are always loaded into

your user address space, normally close below 0x7FFF FFFF. For example, on my system, Kernel32.DLL loads at 0x7C80 0000.

e). Your application may be linked with other DLLs. For example, if your application uses OLE, the OLE DLLs are loaded at 0x5XXX XXXX. If you use other third party DLLs, they are also loaded somewhere in the virtual address space.

Given the items that take your precious virtual address space, only the free holes left are for your application to allocate memory. Obviously, the largest block of virtual address block you can allocate is the largest free hole.

```
FREE: 0x00000000, +0x00010000, end 0x00010000 (total 64 KB)
FREE: 0x00011000, +0x0000f000, end 0x00020000 (total 60 KB)
FREE: 0x00021000, +0x0000f000, end 0x00030000 (total 60 KB)
FREE: 0x00133000, +0x0000d000, end 0x00140000 (total 52 KB)
FREE: 0x00276000, +0x0000a000, end 0x00280000 (total 40 KB)
FREE: 0x002BD000, +0x00003000, end 0x002C0000 (total 12 KB)
FREE: 0x00301000, +0x0000f000, end 0x00310000 (total 60 KB)
FREE: 0x00316000, +0x0000a000, end 0x00320000 (total 40 KB)
FREE: 0x00333000, +0x000cd000, end 0x00400000 (total 820 KB)
FREE: 0x0040B000, +0x7c3f5000, end 0x7C800000 (total 2035668 KB)
FREE: 0x7C8F4000, +0x0000c000, end 0x7C900000 (total 48 KB)
FREE: 0x7C9B0000, +0x02d40000, end 0x7F6F0000 (total 46336 KB)
FREE: 0x7F7F0000, +0x007c0000, end 0x7FFB0000 (total 7936 KB)
FREE: 0x7FFD4000, +0x00009000, end 0x7FFDD000 (total 36 KB)
FREE: 0x7FFDF000, +0x00001000, end 0x7FFE0000 (total 4 KB)
```

Figure 3: Example of Free Virtual Address Blocks without /3GB

```
FREE: 0x00000000, +0x00010000, end 0x00010000 (total 64 KB)
FREE: 0x00011000, +0x0000f000, end 0x00020000 (total 60 KB)
FREE: 0x00021000, +0x0000f000, end 0x00030000 (total 60 KB)
FREE: 0x00133000, +0x0000d000, end 0x00140000 (total 52 KB)
FREE: 0x00276000, +0x0000a000, end 0x00280000 (total 40 KB)
FREE: 0x002BD000, +0x00003000, end 0x002C0000 (total 12 KB)
FREE: 0x00301000, +0x0000f000, end 0x00310000 (total 60 KB)
FREE: 0x00316000, +0x0000a000, end 0x00320000 (total 40 KB)
FREE: 0x00333000, +0x000cd000, end 0x00400000 (total 820 KB)
FREE: 0x0040B000, +0x7c3f5000, end 0x7C800000 (total 2035668 KB)
FREE: 0x7C8F4000, +0x0000c000, end 0x7C900000 (total 48 KB)
FREE: 0x7C9B0000, +0x02d40000, end 0x7F6F0000 (total 46336 KB)
FREE: 0x7F7F0000, +0x007f0000, end 0x7FFE0000 (total 8128 KB)
FREE: 0x7FFE1000, +0x3ffcf000, end 0xBFFB0000 (total 1048380 KB)
FREE: 0xBFFD4000, +0x00006000, end 0xBFFDA000 (total 24 KB)
FREE: 0xBFFDB000, +0x00004000, end 0xBFFDF000 (total 16 KB)
```

Figure 4: Example of Free Virtual Address Blocks with /3GB

I built a very simple C/C++ application that walks through the free holes in the virtual address space using VMAPI function `VirtualQuery()`. (The code is attached in the appendix.) It does not use any fancy DLL that fragments its virtual address space, and its results are shown in Figure 3 and Figure 4.

The two listings show that below 2GB, the largest free block is 2035668KB, or 1988MB. It is smaller than 2048MB (2GB), but actually not too far (only 60MB difference). When /3GB is enabled, the second largest free block is 1048380KB, or 1023MB, starting at 0x7FFE1000. This is the real extra gift that /3GB provides. But the 1988MB and 1023MB are not possible to be combined into a contiguous chunk, due to the stuff that lives in between.

As a side notice, you can also notice that the 1988MB block starts at 0x0040B000. That means my executable roughly occupies from 0x0040 0000 to 0040 B000, which is about 44KB, i.e., a really small program.

Sometimes in an application, not all allocated memory blocks are used equally frequently. The less used memory blocks may be put in paging files if the physical memory is low, and you may only experience little performance degrading, but these memory blocks still use your address space. The result is that your virtual address space runs out so that you cannot allocate new memory, before the physical memory is exhausted. This is the exact reason why the Microsoft Exchange Server suggests /3GB boot option when you have just more than 1GB physical memory.

3.4 DLL Rebasing

In real applications, you can not expect to have an application as small as the free virtual address block walker. That means, more or less, the application may be complex and may need to link to other DLLs.

The DLLs normally do not take a lot of memory space for their code sections. I have never seen DLL files like more than 10MB. However, if your application needs large chunks of contiguous memory, linking to DLLs could give you trouble.

This is because each DLL just like your application has its *base*, which is the preferred location in the virtual address space of your process. If Windows does not find any conflict, i.e., no other code or data is using the same base location, the DLL is loaded there. Now assume you application is linked to a few third party DLLs that use scattered bases, then your virtual address space is soon fragmented and even the largest free hole is not big enough for your application.

Because a lot of parts of a DLL are read only, Windows create file mapping for these parts, so that they are backed by the DLL file[3] instead of the paging files, and the paging file space is saved. The file mapping is injected into the virtual address space at the base

[3] The file mapping is also used for certain parts of the application EXE file when it is launched for running. This is why you can not move, delete, or modify a DLL or EXE file when it is running, because the file is locked as backing of virtual memory.

of the DLL for all processes that use the DLL. Keeping the same base address in all processes is very important for sharing the DLL virtual memory blocks, because when the DLL is built, its code and data are hard coded using the assumed base. Changing the base would have to modify all the spots affected by the assumed base, which are recorded as entries of the relocation table of the DLL. Otherwise, the DLL refers to wrong locations in the process.

If Windows finds out that a DLL cannot be loaded at its preferred base for your process because that location is already used, Windows has to *rebase* the DLL. Rebasing has several consequences. Windows needs to find another location that can accommodate the DLL in your virtual address space. Since it needs to relocate thus modify the DLL contents, it cannot use the DLL file as backing. Windows has to use the paging files as backing for the relocated DLL. It also known that Windows does not share the rebased DLL among processes, probably due to that it is harder to coordinate a new base of the DLL among all processes using it. Even if rebasing reduces the available paging file space, it does not waste your virtual address space. The virtual memory overhead is probably not a big issue if the DLL is not huge. The rebasing can be slow in walking through all the relocation table entries and applying changes, but this probably only happens when the program starts, so it is okay. More concern is on where in the virtual address space Windows puts the DLL. Fortunately, it is reported that Windows tries to find the location down from the 2GB boundary for a DLL to be rebased, so it does not randomly take unused portion of your big free chunk in the virtual address space.

Maybe it is more important to check the DLL file's base, because Windows tries to load the DLL at its preferred base as the first step. The default DLL base is 0x1000 0000 when it is built, just like 0x0040 0000 for an EXE. There is a "base address" option in the linker to change an executable's base to any location in user address space. When there are multiple DLLs used in one application, some people use some schemes to assign different bases to different DLLs to avoid rebasing. This could be a bad thing for your memory hungry application if the DLL bases create very fragmented free blocks in the virtual address space. If your application uses the DLLs with cold links, i.e., the DLL is not loaded programmatically by `LoadLibrary()`, you can use Microsoft Visual Studio's Dependency tool to check the used DLLs of your application, and it also shows the bases of the DLLs.

One of the key subsystem DLLs, Ntdll.DLL, has some hard coded fixed addresses such that it is not even relocatable. Windows never rebase these DLLs, and they have to sit at the exact locations slightly below 2GB in the address space of every process. This contributes to the fact that you can only get around 2GB but not 3GB of free chunk even if you have /3GB enabled.

These are the suggestions to avoid or mitigate the DLL fragmenting free virtual address space. First, avoid using DLLs, use static libraries if possible. This could stop creating some small fragments, for example, the fragment between 0x0040 0000 and 0x1000 0000, which is about 252MB. If you cannot avoid using DLLs, check their bases and maybe

you need to rebuild them with bases that can optimize the layout of your virtual address space for bigger free address chunks.

3.5 Physical Address Extension (PAE)

Getting virtual address space is one side of the problem, and getting physical memory is the other. As you have known, to get more physical memory for your application, you probably need to get more physical memory available for Windows NT. This may not be always true actually, and we will see it soon.

PAE (Physical Address Extension) is a boot option for Windows to enable both itself and the processors to use more address lines to address more than 4GB of physical memory space. When /PAE option is used in booting Windows, the OS breaks the barrier of 4GB and sees more than 4GB of physical memory, if available. For example, when PAE is not enabled and you insert 4GB RAM to your system, some of the RAM goes beyond 4GB due to memory mapped devices and ROM, and is lost. If you enable PAE, this portion of RAM can be used by the system now.

However, PAE is a processor and OS feature. It does not have an API, and it does not break the 4GB virtual address space barrier for each process. If your process does not use AWE, the benefit of PAE is not evident for a single process immediately. PAE enables the OS to use more physical memory, thus the set of processes can use more physical memory, so any single process may be less subject to page faults and the benefits are really here.

PAE also comes at a cost. Because PAE needs extra address bits, the Page Directory Entry (PDE) and page table entry (PTE) to describe a (4KB) memory page are larger. Non-PAE system uses 4-byte PDEs and PTEs, but PAE system uses 8-byte PDEs and PTEs. The result is that a 4KB page now has 512 PDEs or PTEs, instead of 1024 PDEs or PTEs. This means that Windows has to use more pages, i.e., memory, to manage the data pages that are available to processes. This situation exacerbates when /3GB is used, because the kernel has tighter memory space for bigger page directories and page tables. Table 2 shows that when /3GB is used together with /PAE, the OS can only make use of at most 16GB physical memory space. Taking out the memory mapped devices and ROM, the OS can only address slightly less than 16GB physical memory.

Table 2: /PAE and /3GB Effects on 32-bit Windows

Boot options		*Physical address space available to OS*	*Virtual address space per user process*
		4GB	2GB
/3GB		4GB	3GB
	/PAE	64GB	2GB
/3GB	/PAE	16GB	3GB

3.6 Conclusion

For 32-bit applications, there are a few factors affecting the memory that an application can use. We assume the application uses the CRT heap for its memory management in most cases.

Unless you use non-portable and mapping/remapping based file mapping or AWE, your application process cannot access more than 2GB memory, no matter it is virtual memory or physical memory, due to virtual memory space limitation. By using /3GB and/or /USERVA boot options, this limitation can be relaxed to at most 3GB. You may also want to check the DLL bases to guard against excessive virtual address space fragmentation.

You can add as much as physical memory to the system and increase the physical memory available to the OS, but it does not increase the physical memory available to the application process beyond the 2GB/3GB virtual address limitation. Your application can get at most 2GB/3GB physical memory given there is enough physical memory, after deducting the physical memory used by the operating itself and other processes. Because the memory your application process directly gets is virtual memory, Windows has full power of putting your data/code in paging files at its own discretion. By unloading unnecessary system components, such as disabling devices and stopping services not in use, and closing unnecessary applications, your application has more chance to have its code and data in physical memory up to 2GB/3GB upper limit.

4. Memory on 64-bit PC

As we have already seen, to deal with applications requiring large memory is very tricky in 32-bit Windows, due to the virtual address space and probably also windows physical address space limitations. Fortunately, both 64-bit processors and 64-bit Windows are out to help. In a 64-bit system, the limitation is obviously not the address space, because it has 16EB (Exa Bytes), i.e., 2^64 bytes, or 16 Giga GB, or 16Mega TB, in theory.

In the PC world, the 64-bit architecture is an extension to the 16-bit and 32-bit x86, and is called x86-64, or x64 in short. x64 includes AMD's AMD64 and Intel's Intel64 (the two have only minor differences). Although Microsoft also has some Windows versions supporting Intel's Itanium 64-bit processors (IA64), IA64 was not designed for desktop/workstation applications and clearly x64 will be the one that replaces the current 32-bit x86 architecture.

There is no application that would require 16EB memory in the near future, so current x64 CPUs do not really have 64 address pins to have 64-bit physical addresses. For example, the first x64 CPU, AMD Opteron, implements 40-bit physical address (1TB) and 48-bit virtual address (256TB). New AMD64 CPUs implement 52-bit physical (4PB, Peta Bytes) address and 64-bit virtual address (16EB). Initial Intel64 processors had 36-bit physical address (64GB), and the new ones have 40-bit physical address (1TB).

There are two categories of Windows x64 editions available: Windows XP/Vista x64 Editions and Windows Server 2003 x64 Editions, for desktop and server systems respectively. Windows x64 uses 44-bit virtual address (16TB). The lower half of that, 8TB (from 0x0000 0000 0000 0000 to 0x0000 07FF FFFF FFFF) is available for each user mode process; and the upper 8TB (from 0xFFFF F800 0000 0000 to 0xFFFF FFFF FFFF FFFF) is reserved for the kernel. There is a huge gap between the two 8TB blocks in the 64-bit virtual address space. This area is for the user address to grow up in and kernel address to grow down in on future Windows editions, without disrupting backward system compatibility. The physical memory sizes supported are different for different Windows x64 editions, as listed in Table 3.

Table 3: Maximum RAM Supported by Windows x64 Editions

Windows x64 Editions	*Maximum RAM Supported*
Windows XP Professional x64	128GB
Windows Vista Home Basic x64	8GB
Windows Vista Home Premium x64	16GB
Windows Vista Business x64	128+GB
Windows Vista Enterprise x64	128+GB
Windows Vista Ultimate x64	128+GB
Windows Server 2003 Standard x64	32GB
Windows Server 2003 Enterprise x64	1TB
Windows Server 2003 Datacenter x64	1TB

Windows x64 is backward compatible with Win32 applications, so your Win32 application can still run under Windows x64. There is also a very nice feature of Windows x64. If your Win32 application is linked with /LARGEADDRESSAWARE flag, the user mode virtual address space is neither 2GB, nor 3GB, but *full* 4GB. Windows x64 just put its kernel completely out of the 4GB address space of a Win32 process. This is a free gift of Windows x64 without you rewriting anything.

To fully leverage the power of 64-bit systems, you need to write your application in 64-bit. This opens the door to almost unlimited memory. (Well, ironically, I remember that people said similar things when they moved from 16-bit to 32-bit, when 4GB memory was not imaginable and everybody was using 4MB of memory or so.) You are no longer limited by the address space at least in near future with 64-bit Windows, and adding physical memory really improves your application's performance much directly.

Once the address space barrier is broken, the real limitation is actually from the physical memory and the circuitry supporting physical memory. First of all, high density DDR or DDR2 SDRAM is still very expensive, and not many can afford tens of GBs of memory right now. Second, the motherboard/chipset often puts limitations on physical memory capacity. It is not surprising that you can see entry-level x64 PCs that only support a maximum of 4GB physical memory. I do not recommend these 64-bit systems, because its memory extensibility is crippled. If you are looking for 64-bit systems, check at least

the maximal supported memory size, make sure you are comfortable with it, no matter it is 8GB, 16GB, or more.

4. Conclusion

Due to address space limitations, applications on x86 32-bit Windows cannot access more than 2GB memory (3GB if /3GB boot option is enabled), unless it uses some dirty paging techniques to bring in the larger virtual or physical memory into the limited virtual address space. Portable applications using the CRT heap to obtain memory are specifically confined by the 2GB/3GB address space barrier. Since other objects such as DLLs may also take the virtual address blocks and fragment the virtual address space, the actual memory that an application can get is smaller, so is the largest contiguous memory chunk. Adding physical memory to the system, and enabling PAE if needed, can increase the physical memory available to the operations system, but that only increases the chance that a single application gets physical memory instead of virtual memory backed on disk paging files.

When an application needs more than 2GB/3GB memory, moving to x64 systems is the obvious choice. The virtual address space limit is completely lifted, and the problem mostly boils down to find a system in which the both the hardware and the operating system support big enough physical memory.

References:

[1] Microsoft Developer Network (MSDN): http://msdn.microsoft.com
[2] Microsoft Knowledge Base: http://support.microsoft.com/search/?adv=1
[3] The Old New Things: http://blogs.msdn.com/oldnewthing/
[4] The Wikipedia: http://www.wikipedia.org/

Appendix: The Free Virtual Address Block Walker

```
// Binglong Xie, 2007.
// link with /LARGEADDRESSAWARE option to make sure
// this application tests regions above 2GB.

#include <windows.h>
#include <stdio.h>
#include <assert.h>

const unsigned long KB = 1024UL;
const unsigned long MB = KB * KB;
const unsigned long GB = MB * KB;
void WalkFreeRegions(DWORD start_address, DWORD length);

int main(int, char* [])
{
    WalkFreeRegions(0UL, (DWORD)(-1));
    return 0;
}

void WalkFreeRegions(DWORD start_address, DWORD length)
{
    if( length==0 )
        return;
    const unsigned long VM_granularity = 4*KB;
    DWORD addr0 = start_address/VM_granularity*VM_granularity;
    DWORD addr1 = start_address + length;
    if( addr1<addr0 )    // overflow. Use max address
        addr1 = (DWORD)(-1);
    addr1 = addr1/VM_granularity*VM_granularity;
    printf("Finding free virtual address regions in [0x%8.8lX, 0x%8.8lX]"
           "\nLength=0x%8.8lX (%ld KB)\n",
           addr0, addr1, addr1-addr0, (addr1-addr0)/KB);
    MEMORY_BASIC_INFORMATION mbi;
    bool no_info = false;
    while( addr0<=addr1 )    {
        DWORD addr = addr0;
        DWORD bytes = VirtualQuery((BYTE*)addr, &mbi, sizeof(mbi));
        if( bytes==0 )        {   // no info available.
            if( !no_info )
            {
                printf(" NONE: 0x%8.8lX VirtualQuery() has no info START\n", addr);
                no_info = true;
            }
            addr += VM_granularity;
        }
        else {
            if( no_info ) {   // recovered from no_info
                printf(" NONE: 0x%8.8lX VirtualQuery() has no info END\n", addr);
                no_info = false;
            }
            if( mbi.State==MEM_FREE ) {   // free region.
                printf(" FREE: 0x%8.8lX, +0x%8.8lx, end 0x%8.8lX (total %ld KB) \n",
                addr, mbi.RegionSize, addr+mbi.RegionSize, mbi.RegionSize/KB);
            }
            else {   // non-free region
            }
            addr += mbi.RegionSize;
        }
        if( addr<addr0 )  // overflow
            break;
        addr0 = addr;
    }
    if( no_info )  {
        printf(" NONE: 0x%8.8lX VirtualQuery() has no info END\n", addr0);
        no_info = false;
    }
}
```