HeapManager
or
a Multi-Thread Garbage Collector

2003-10-25 07:41

by Thomas Weise

tweise@gmx.de
http://www.tu-chemnitz.de/~weist/it/hm/index.htm

Preface

Today, a program's performance weights more than it's memory consumption. For a server application for example, it is most important to server many clients. The faster the algorithms run, the faster a client can be served, the more clients can be handled at a time (of course, if enough resources are available).
When creating a new server application, one should have an eye on performance of each one of it's parts.
One part is the memory management, the memory allocation, de-allocation and re-allocation.
These operations are normally done by the heap functions in a Windows system.
The Windows heap functions are thread-safe, but with some tricks one could increase performance of them.
Using these operations in a new scheme, this paper tries to show a way on how that can be done.

A garbage collector for multi-thread applications is to be introduced which allows quasi-simultaneous memory operations.

I don't know how far this technique is already used or if it is really new, however, it can often grant a performance boost of about five times (Windows 2000) or two times (Windows XP) of the speed of normal memory allocation and should never (worst case) be noticeable slower.

The algorithms are tested under Microsoft Windows XP and Microsoft Windows 2000, but should work on every OS, which supports standard memory operations and thread-handling routines.

Index

1.	**Basic Ideas**	4
1.1	Garbage Collector	4
1.2	Fast Access	4
1.3	Simultaneous Access	5
2.	**Test Results**	6
3.	**Source Code**	8
3.1	Basics.h	8
3.2	Basic_Memory_Manager.h	12
3.3	Basic_Memory_Manager.cpp	13
3.4	Test.cpp	22

1 Basic Ideas

1.1 Garbage Collector

One very important part of the heap manager is the garbage collector. It is placed between the OS-defined allocation/de-allocation/re-allocation routines and is used to keep memory once allocated for a defined time in a hash if it is de-allocated.
An application often needs buffers or big objects. Whenever you allocate memory, system spends time in kernel mode, does a lot of stuff searching a free block, initializes data to remember the block it grants and so on. If you free a block of memory, everything is done the other way round.
Now, if one frequently allocates and de-allocates memory, this causes a lot of work for the OS.
So if a memory block freed once can be kept in a list for a while, to see if it can be used again, one could possible save some time.
Of course, while the memory block is "kept alive" and is not yet re-used, memory is wasted. This is the trade-off for a higher performance.

The garbage collector implemented works on this principle, it keeps freed blocks alive for between two and four seconds per default.
However, not all memory blocks are collected like this. Blocks smaller than 64 Byte or bigger than 64 mega bytes are not kept, because either it is not worth the additional logic, it would occur too seldom or the wasted memory would be too much.

1.2 Fast Access

However, the access to the garbage collector must be faster than normal allocation, or else it would make no sense.

The first problem is the data structure used.
An unsorted sequential list would be easy to maintain but slow in access, because on worst case one would search through the whole list until discovering that no fitting memory block is cached.
A sorted sequential list would be fast to access (one could use binary search with O(log n)), but slow to maintain, because every en-queuing of freed blocks would cause a movement of (worst case) the whole list one item back.
Using a Heap would cause a somewhat recursive searching and O(log n) inserting algorithm.

A good solution for this problem would be hashing.
A very common way for handling for memory block management is the "buddy processing", which will be used in the HeapManager.
It provides a very simple hash function, which only needs two lines of assembler.

```
    bsr eax, assize

#if MIN_MEM > 0
    sub eax, MIN_MEM
#endif
```

The classification criteria will be the most significant bit. Every block enqueued in the hash will be classified by it's most significant bit, meaning that sizes like 256, 325, 457 and 511 will belong to one class while 131'072, 161'072 and 231072 belong to another one.
En-queuing operations will add new blocks always at the end of the hash.
Searching for a free block of a specified size will be done in a simple fashion. First, the size is classified. Then, the hash list of that class will be searched sequential for a block *bigger or equal* the searched size. On the first glance, this looks like a big waste. But in the worst case, the block found is twice as big as the smallest block that *could possible* be found (worst case). However, this searching algorithm leads to a proper result in the halve time in average case.

1.3 Simultaneous Access

The usage of the hash described on the previous page yet has another big advantage.
If one thread allocates memory from the operating system in a multi-thread application (with mutual exclusion turned on, of course), all other threads trying to allocate, free or re-allocate memory at the same time are blocked, meaning put into a wait state.
By classifying the memory block sizes, the memory manager can lock a range of memory instead of the whole hash.
Coming back to the example, if one thread tries to allocate a memory block of 256 bytes, no other thread can simultaneously allocate, free or re-allocate a block of 256, 325, 457 or 511 bytes, while it is no problem to allocate 131’072, 53’452 or 1024 bytes.
By that, a maximum of 21 allocations at a time per default are possible with the memory manager.

```
typedef         struct          {
                dword           length;
                dword           count;
                event           lock;
                p_block         blocks;
                                } range;
```

As mutual exclusion objects, events are used.

2 Test Results

The memory manager was tested on two different systems. The output shows the time spent in the tests, memory usage was determined with the task manager.
The test program was implemented using Microsoft Visual Studio 6.0 Service Pack 5 Visual C++, which was a gift of Microsoft Academic Alliance.

With twenty simultaneous threads running, test were done with many different loop lengths. As an example, two of the output windows are shown below.

D:\Programming\HeapManager\Re...

```
HeapManager                 : 657.865965 s.
HeapAlloc                   : 3919.445886 s.
GlobalAlloc                 : 4409.730882 s.
malloc                      : 1391.741226 s.
HeapManager vs. HeapAlloc   : 1:5.957818
HeapManager vs. GlobalAlloc: 1:6.703084
HeapManager vs. malloc      : 1:2.115539
```

2.1 20 Threads, Microsoft Windows XP notebook system with 512 MB RAM , Intel 2 GHz Pentium 4 CPU

E:\HeapManager.exe

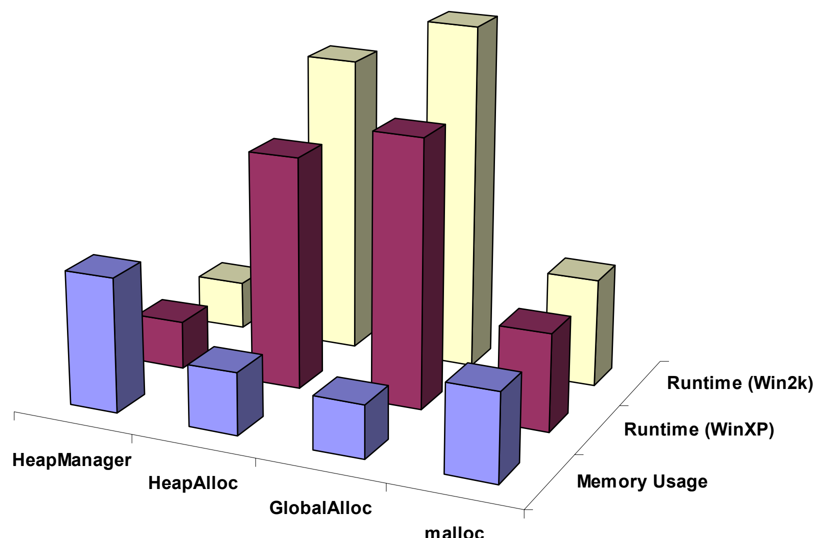
```
HeapManager                 : 2509.959144 s.
HeapAlloc                   : 15524.052491 s.
GlobalAlloc                 : 17995.776656 s.
malloc                      : 5628.022699 s.
HeapManager vs. HeapAlloc   : 1:6.184982
HeapManager vs. GlobalAlloc: 1:7.169749
HeapManager vs. malloc      : 1:2.242277
```

2.2 20 Threads, Microsoft Windows 2000 stationary system, 224 MB RAM, Intel 450 MHz Pentium 2 CPU

The interesting thing is, that the ratio between HeapManager/malloc to HeapAlloc/GlobalAlloc changes with the length of the tests. On short tests, HeapAlloc/GlobalAlloc are faster while using less memory. The longer the tests, the more allocations/de-allocations done, the more kicks the performance gaining effect in. To get stable results, very long tests were run.
Therefore, the results are only representative for systems that run a long time, 10 hours, for example.

The most memory is used by the HeapManager, while it is also the fastest of these four test cycles.
On both systems, the memory requirements were somewhere between 1.5 and 2.2 times the normal requirements, which is, of course, a lot.

By the way, malloc seems also to implement a way of garbage collecting or something else to increase performance. It is remarkable faster than HeapAlloc/GlobalAlloc, but needs also more memory.



2.3 20 Threads, memory/runtime diagram

With 40 threads the results are even more interesting.

D:\Schule\Anderes (S, 1-5)\Prosemi...

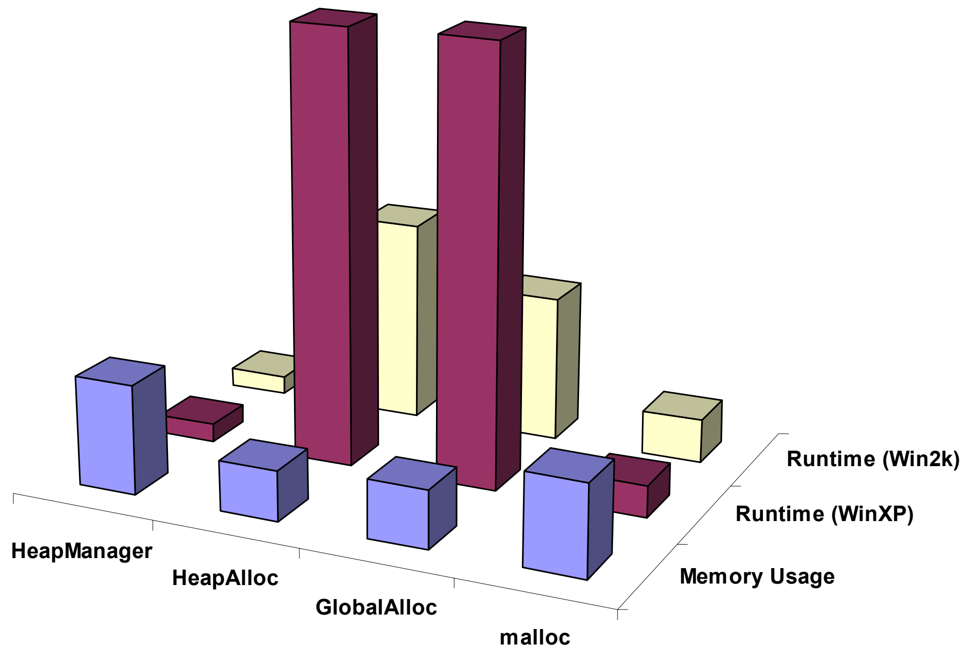
```
HeapManager                  : 588.836706 s.
HeapAlloc                    : 14024.796667 s.
GlobalAlloc                  : 14806.540760 s.
malloc                       : 1028.228520 s.
HeapManager vs. HeapAlloc    : 1:23.817803
HeapManager vs. GlobalAlloc: 1:25.145411
HeapManager vs. malloc       : 1:1.746203
```

2.4 40 Threads, Microsoft Windows XP notebook system with 512 MB RAM , Intel 2 GHz Pentium 4 CPU

E:\HeapManager.exe

```
HeapManager                  : 2023.539707 s.
HeapAlloc                    : 22279.586472 s.
GlobalAlloc                  : 16075.665672 s.
malloc                       : 4817.987926 s.
HeapManager vs. HeapAlloc    : 1:11.010205
HeapManager vs. GlobalAlloc: 1:7.944329
HeapManager vs. malloc       : 1:2.380970
```

2.5 40 Threads, Microsoft Windows 2000 stationary system, 224 MB RAM, Intel 450 MHz Pentium 2 CPU



2.6 40 Threads, memory/runtime diagram

3 Source Code

As stated above, the source code for the heap manager is written for a Microsoft Visual C++ ® 6.0 Service Pack 5 Environment. It imports all Windows API it needs by itself.
The HeapManager is basically one dynamic link library (dll) to be imported by a program.
Make sure that Project-Settings-Link-General-Object/Library Modules is set to *"mpr.lib ws2_32.lib kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib"* if you want to implement it on your own.
Although the source is an example only for Windows, one could easily implement it for Linux or whatever for sure.

3.1 Basics.h

This header is used to import all Windows API, constants and data types required.

```
#ifndef  BASICS_USED
#define  BASICS_USED

//configuration

#pragma  inline_depth(255)
#pragma  inline_recursion(on)
#pragma  auto_inline(on)
#pragma  optimize("gpty", on)
#pragma  pointers_to_members(full_generality, virtual_inheritance)


#pragma warning (disable : 4100)
#pragma pack (push)
#pragma pack (1)


//shortcuts
#define SC       __stdcall
#define FC       __fastcall

#define IMP      __declspec(dllimport)

#define NULL        0
#define MAX_DWORD   0xffffffff
#define DWORD_MSB   0x80000000


//macros
#define DEFINE_TYPES(a)    typedef const a    c_##a;     \
                           typedef a*         p_##a;     \
                           typedef const p_##a          cp_##a;    \
                           typedef c_##a*     pc_##a;    \
                           typedef a&         r_##a;     \
                           typedef p_##a*     pp_##a;    \

#define DEFINE_TYPE(a, b)  typedef b          a;         \
                           DEFINE_TYPES(a)


#define RCAST(a, b)              (reinterpret_cast<a>(b))
```

```
#define ERROR_UNKNOWN_MEMORY     8
#define ERROR_OUTOFMEMORY        14

//com constants
#define S_OK                NULL
#define S_FALSE             0x00000001
#define E_ERRORBASE         DWORD_MSB
#define E_UNEXPECTED        (E_ERRORBASE | 0x0000FFFF)
#define E_OUTOFMEMORY       (E_ERRORBASE | 0x0007000E)
#define E_POINTER           (E_ERRORBASE | 0x00004003)
#define E_FAIL              (E_ERRORBASE | 0x00004005)


//thread constants
#define    THREAD_BASE_PRIORITY_MAX   2

//thread constants
#define    THREAD_PRIORITY_LOWEST          THREAD_BASE_PRIORITY_MIN
#define    THREAD_PRIORITY_BELOW_NORMAL    (THREAD_PRIORITY_LOWEST+1)
#define    THREAD_PRIORITY_NORMAL          NULL
#define    THREAD_PRIORITY_HIGHEST         THREAD_BASE_PRIORITY_MAX
#define    THREAD_PRIORITY_ABOVE_NORMAL    (THREAD_PRIORITY_HIGHEST-1)
#define    THREAD_PRIORITY_ERROR_RETURN    (MAX_DWORD)
#define    THREAD_PRIORITY_TIME_CRITICAL THREAD_BASE_PRIORITY_LOWRT
#define    THREAD_PRIORITY_IDLE            THREAD_BASE_PRIORITY_IDLE


//status constants
#define    STATUS_WAIT_0     NULL
#define    STATUS_USER_APC   0x000000C0


//wait constants
#define    INFINITE             MAX_DWORD//0xFFFFFFFF
#define    WAIT_OBJECT_0        (((STATUS_WAIT_0 ) + NULL ))
#define    WAIT_IO_COMPLETION   (STATUS_USER_APC)


//memory constants
#define    GMEM_FIXED       NULL
#define    GMEM_MOVEABLE    2


DEFINE_TYPE (          word,        unsigned __int16)
DEFINE_TYPE (         dword,        unsigned __int32)
DEFINE_TYPE (         qword,        unsigned __int64)
DEFINE_TYPE (       integer,        signed __int32)
DEFINE_TYPE (       hugeint,        signed __int64)
DEFINE_TYPE (       unichar,        unsigned short)
DEFINE_TYPE (     unistring,        unichar*)
DEFINE_TYPE (     asciichar,        char)
DEFINE_TYPE (   asciistring,        asciichar*)
DEFINE_TYPE (       boolean,        dword)
DEFINE_TYPE (       pointer,        void*)
DEFINE_TYPES(         float)
DEFINE_TYPES(        double)

DEFINE_TYPE (        handle,        dword)
DEFINE_TYPE (         event,        handle)
DEFINE_TYPE (        thread,        handle)
DEFINE_TYPE (          heap,        handle)
DEFINE_TYPE (waitable_timer,        handle)
DEFINE_TYPE (      hresult,       dword)
```

```
typedef      struct     {
   dword      length;          // sizeof(MEMORYSTATUS)
   dword      memory_load;      // percent of memory in use
   dword      total_phys;       // bytes of physical memory
   dword      avail_phys;       // free physical memory bytes
   dword      total_pagefile;   // bytes of paging file
   dword      avail_pagefile;   // free bytes of paging file
   dword      total_virtual;    // user bytes of address space
   dword      avail_virtual;    // free user bytes
              } memorystatus;
DEFINE_TYPES (memorystatus)

DEFINE_TYPE  (filetime,          qword)

typedef       struct    {
    word      year;
    word      month;
    word      dayofweek;
    word      day;
    word      hour;
    word      minute;
    word      second;
    word      milliseconds;
              } systemtime;
DEFINE_TYPES (systemtime)


// SECURITYATTRIBUTES type
typedef          struct    {
      dword     length;
      pointer   security_descriptor;
      boolean   inherithandle;
              } security_attributes;
DEFINE_TYPES    (security_attributes)


extern "C" {

IMP c_dword     SC GetLastError       (void);

IMP c_pointer   SC GlobalAlloc        (c_dword         aflags,
                                       c_dword         asize);
IMP c_pointer   SC GlobalReAlloc      (c_pointer       amem,
                                       c_dword         asize,
                                       c_dword         aflags);
IMP c_pointer   SC GlobalFree         (c_pointer       amem);

IMP c_heap      SC GetProcessHeap     (void);
IMP c_pointer   SC HeapAlloc          (c_heap          aheap,
                                       c_dword         aflags,
                                       c_dword         asize);
IMP c_pointer   SC HeapReAlloc        (c_heap          aheap,
                                       c_dword         aflags,
                                       c_pointer       amem,
                                       c_dword         abytes);
IMP c_boolean   SC HeapFree           (c_heap          aheap,
                                       c_dword         aflags,
                                       c_pointer       amem);
IMP c_dword     SC HeapSize           (c_heap          aheap,
                                       c_dword         aflags,
                                       c_pointer       amem);
```

```
IMP void      SC GlobalMemoryStatus(r_memorystatus astatus);
IMP void      SC RtlMoveMemory      (c_pointer      adestination,
                                     c_pointer      asource,
                                     c_dword        asize);
IMP void      SC RtlZeroMemory      (c_pointer      amemory,
                                     c_dword        asize);

IMP c_waitable_timer SC CreateWaitableTimerW(
                                     pc_security_attributes asecurity,
                                             c_boolean     amanualreset,
                                             c_unistring   aname);
IMP c_event   SC CreateEventW       (pc_security_attributes asecurity,
                                     c_boolean         amanualreset,
                                     c_boolean         ainitialstate,
                                     c_unistring       aname);
IMP c_boolean SC SetEvent           (c_event           aevent);
IMP c_boolean SC SetWaitableTimer   (c_waitable_timer atimer,
                                     cp_filetime       aduetime,
                                     c_dword           aperiod,
                                     c_pointer         acompletion,
                                     c_dword           acompletionarg,
                                     c_boolean         aresume);
IMP c_dword   SC WaitForSingleObject(c_handle           ahandle,
                                     c_dword            amilliseconds);
IMP c_dword   SC WaitForSingleObjectEx(c_handle           ahandle,
                                       c_dword            amilliseconds,
                                       c_boolean          aalertable);
IMP void      SC GetSystemTimeAsFileTime(r_filetime         afiletime);
IMP c_boolean SC CloseHandle             (c_handle           ahandle);

IMP c_thread  SC CreateThread       (p_security_attributes asecurity,
                                     c_dword           astacksize,
                                     c_pointer         astartaddress,
                                     c_pointer         aparameter,
                                     c_dword           acreationflags,
                                     r_dword           athreadid);

IMP c_boolean SC SetThreadPriority  (c_thread          athread,
                                     c_dword           apriority);

IMP void      SC Sleep              (c_dword           amilliseconds);

IMP c_boolean SC IsBadReadPtr       (c_pointer         apointer,
                                     c_dword           asize);
IMP c_boolean SC IsBadWritePtr      (c_pointer         apointer,
                                     c_dword           asize);

IMP c_dword   SC WaitForMultipleObjects(c_dword            acount,
                                       pc_handle          ahandles,
                                       c_boolean          awaitall,
                                       c_dword            amilliseconds);

  }

#define MAINA     main     (c_dword         aargc,  \
                            c_asciistring   argv[], \
                            c_asciistring   aenvp[])


#pragma pack (pop)
#endif
```

3.2 HeapManager.h

Here we have the routines exported by HeapManager.cpp.

```
#ifndef         HEAPMANAGER_USED
#define         HEAPMANAGER_USED


#include        "Basics.h"


c_hresult        init_memory_management       (void);
c_hresult        close_memory_management      (void);

c_hresult     FC error                        (void);
c_hresult     FC move                         (c_pointer      adest,
                                               c_pointer      asource,
                                               c_dword        asize);

c_dword       FC max_ram                      (void);
c_dword       FC max_virtual                  (void);
c_dword       FC free_ram                     (void);
c_dword       FC free_virtual                 (void);


c_hresult     FC allocate                     (c_dword        asize,
                                               r_pointer      amem);
void          FC free                         (r_pointer      amem);
c_hresult     FC reallocate                   (c_dword        anewsize,
                                               r_pointer      amem);
c_dword       FC size                         (c_pointer      amem);


#endif
```

3.3 HeapManager.cpp

In this source code file, the heap manager is implemented.
A set of behavioural constants defines how the memory manager works.
MIN_MEM and MAX_MEM define the lowest and highest MSB (most significant bit) which is set on all memory blocks managed, so by default (MIN_MEM = 6, MAX_MEM = 26), only block sizes between 64 and 64 mega bytes are managed. For other sized blocks, the standard memory routines are used.
With INTERVALL the interval times are defined on which the cleaner thread becomes “active” and with TTL (time to live) a rule for the count of interval is set that a block can keep in the cache.

```
#include        "Basics.h"


c_hresult  FC translate_error                (c_dword        awinerror)
  {
  if(awinerror)
    {
    switch(awinerror)
      {
      case ERROR_UNKNOWN_MEMORY                : return E_POINTER;
      case ERROR_OUTOFMEMORY                   : return E_OUTOFMEMORY;
      default                                  : return E_FAIL;
      }
    }
  return S_OK;
  }


c_hresult  FC error                          (void)
  {
  hresult retval = translate_error(GetLastError());
  if(retval == S_OK) retval = E_FAIL;
  return retval;
  }


c_hresult  FC worst                              (c_dword        acomerror1,
                                                  c_dword        acomerror2)
  {
  if(acomerror2 > acomerror1) return acomerror2;
  return acomerror1;
  }

//defines global parameters for our memory management
#define         MIN_LIST_LENGTH    100
#define         INTERVALL          2000
#define         TTL                3


//defines the flags for the heap operations
#define         H_RA_F             0
#define         H_RA2_F            HEAP_REALLOC_IN_PLACE_ONLY
#define         H_A_F              0
#define         H_F_F              0
#define         H_S_F              0
```

behavioural constants

```
//defines the range that is managed
#define     MIN_MEM          6
#define     MAX_MEM          26
#define     MEM_LEN          ((MAX_MEM - MIN_MEM) + 1)
#define     MIN_MANAGED       (1 << MIN_MEM)
#define     MAX_MANAGED      ((1 << (MAX_MEM + 1)) - 1)

//normalize and denormalize pointers
#define     ORIG_PTR(p)     RCAST(cp_dword,   (RCAST(c_dword, p) - 4))
#define     NORM_PTR(p)     RCAST(c_pointer,  (RCAST(c_dword, p) + 4))
```

```
//now here come the data structures used
typedef          struct          {
                 p_dword         ptr;
                 integer         ttl;
                               } block;

#pragma pack(push)
#pragma pack(1)
DEFINE_TYPES(block);
#pragma pack(pop)

//and here we got our main data structure
typedef          struct          {
                 dword           length;
                 dword           count;
                 event           lock;
                 p_block         blocks;
                                 } range;


//these are the internal variables
range            memory[MEM_LEN];
waitable_timer   timer        = 0;
memorystatus     ms;
heap             mainheap     = 0;
thread           threadhandle = 0;
dword            threadid     = 0;


#pragma warning (push)
#pragma warning (disable : 4035)
__forceinline c_dword classify(c_dword asize)
  {

//the highest bit indicates the block-list that the size belongs to
//for 1024, 1789, 2000 it would be (10 - MIN_MEM)
//for 131072, 151121 it would be (17 - MIN_MEM) and so on

    __asm
      {
      bsr eax, asize

#if MIN_MEM > 0
      sub eax, MIN_MEM
#endif

      }
  }
#pragma warning (pop)
```

```
__forceinline c_hresult shrink(c_dword arange)
  {

//we want to resize the block-lists as seldomly as possible
//so therefore ease the criteria

  if(   (memory[arange].count       >= MIN_LIST_LENGTH)  &&
       ((memory[arange].count * 3)  <  memory[arange].length ) )
    {
    pointer p = HeapReAlloc(mainheap, H_RA_F, memory[arange].blocks,
                            sizeof(block) * memory[arange].count);
    if(p)
      {
      memory[arange].length = memory[arange].count;
      }
    else return error();
    }

  return S_OK;
  }


__forceinline c_boolean wait_for(c_handle ahandle)
  {
  dword i;
//allow io-completion returns, but strictly check for object_0

  while ((i = WaitForSingleObjectEx(ahandle, INFINITE, true)) ==
                                                WAIT_IO_COMPLETION) ;

  return (i == WAIT_OBJECT_0);
  }


__forceinline c_pointer find(c_dword asize)
  {
  if((asize >= MIN_MANAGED) && (asize <= MAX_MANAGED))
    {
    dword   rng = classify(asize);

    dword   max = rng + 4;            //blocks with more than 8 times the
    if(max > MEM_LEN) max = MEM_LEN;  //searched size base shall not be regarded
                                      //because it would be a waste

    dword   i;
    p_dword retval;

  //also recognize that we just be blocking one size-sector at a time,
  //so other threads can still allocate memory with sizes of less than
  //the half or more than the double of our size at the same time
  //the windows heap routines instead block the whole heap,
  //so we can be faster in multi-thread apps

    while((rng  < max) && wait_for(memory[rng].lock))
      {

      for(i = 0; i < memory[rng].count; i++)
        {
        if( (retval = memory[rng].blocks[i].ptr)[0] >= asize)
          {
          memory[rng].blocks[i] = memory[rng].blocks[--memory[rng].count];
          SetEvent(memory[rng].lock);
          return NORM_PTR(retval);
```

```
          }
        }

      SetEvent(memory[rng].lock);
      rng++;
      }
    }

  return NULL;
  }


__forceinline void enqueue(p_dword aptr)
  {
  dword   s = aptr[0];
  boolean b = false;

//here we again just lock one size range

  if((s >= MIN_MANAGED) && (s <= MAX_MANAGED))
    {
    s      = classify(s);

    if(wait_for(memory[s].lock))
      {
      dword c, l;

      if( (c = (memory[s].count++)) >= (l = memory[s].length))
        {
        l++;
//by letting the block list grow fast, we avoid frequent reallocation
        pointer p = HeapReAlloc(mainheap, H_RA_F, memory[s].blocks,
                                 (2 * l) * sizeof(block));
        if(p) l *= 2;
        else
          {
          p = HeapReAlloc(mainheap, H_RA_F, memory[s].blocks,
                          l * sizeof(block));
          }
        if(p)
          {
          memory[s].length = l;
          memory[s].blocks = RCAST(p_block, p);
          b                = false;
          }
        else
          {
          memory[s].count  = c;
          goto setev;
          }
        }

      memory[s].blocks[c].ptr = aptr;
      memory[s].blocks[c].ttl = TTL;

setev:
      SetEvent(memory[s].lock);
      }
    }
  //if enqueueing failed, we free the block
  if(b) HeapFree(mainheap, H_F_F, aptr);
  }
```

```
c_dword        SC cleaner_thread              (c_pointer aparameter)
  {
  dword   i, k;

//again we just be blocking one small section of the memory aggregation
//note that this is a little bit slower in single thread apps than the
//waitformultipleobjects on all locks, but faster on multi threadding

//our thread gets only active every INTERVALL milliseconds
  while(WaitForSingleObject(timer, INFINITE) == WAIT_OBJECT_0)
    {
    for(i = 0; i < MEM_LEN; i++)
      {
      if(WaitForSingleObject(memory[i].lock, INFINITE) != WAIT_OBJECT_0)
return 0;

      k = 0;

      while(k < memory[i].count)
        {
    //when it's time to free the block, meaning it's time to live ellapsed
        if( (--memory[i].blocks[k].ttl) <= 0 )
          {
          HeapFree(mainheap, H_F_F, memory[i].blocks[k].ptr);
    //we exchange it with the last block in the list
          memory[i].blocks[k] = memory[i].blocks[--memory[i].count];
          }
        else k++;
        }
    //now we shrink the list down
      shrink(i);
    //and allow further operations
      SetEvent(memory[i].lock);
      }

    }

  return 0;
  }


c_hresult        init_memory_management      (void)
  {
  hresult retval = S_OK;

  if(mainheap == 0)
    {
    GlobalMemoryStatus(ms);
    RtlZeroMemory(&memory, sizeof(memory));
    timer          = 0;
    threadhandle = 0;
    threadid       = 0;

    mainheap = GetProcessHeap();

    if(mainheap)
      {

      dword i;
      for(i = 0; (i < MEM_LEN) && (retval == S_OK); i++)
        {
        memory[i].lock = CreateEventW(NULL, false, true, NULL);
        if(memory[i].lock)
```

```
          {
          memory[i].blocks = RCAST(p_block, HeapAlloc(mainheap, 0,
                                  MIN_LIST_LENGTH * sizeof(block)));
          if(memory[i].blocks) memory[i].length = MIN_LIST_LENGTH;
          else                 retval = error();
          }
        else retval = error();
        }

      if(retval == S_OK)
        {
        timer = CreateWaitableTimerW(NULL, false, NULL);

        if(timer)
          {
          filetime time;
          GetSystemTimeAsFileTime(time);
          time += INTERVALL * 10000;

          if(SetWaitableTimer(timer, &time, INTERVALL, NULL, 0, false))
            {
            threadhandle = CreateThread(NULL, 512, &cleaner_thread, 0,
                                        0, threadid);

            if(threadhandle)
              {
//we can increase performance by giving the cleaner thread a high priority
//so every time it becomes active, it runs fast
//it only becomes active every INTERVALL milliseconds, so we don't disturb
//the other threads too often
              SetThreadPriority(threadhandle,
                                        THREAD_PRIORITY_ABOVE_NORMAL);
              goto ende;
              }

            }
          }

        retval = error();
        }

      }
    else retval = error();

    }

ende:

  return retval;
  }
```

```
c_hresult         close_memory_management      (void)
  {
  hresult retval = S_OK;
  dword   i, j;

  //by killing the timer and the locks, WaitForSingleObject will not return
  //WAIT_OBJECT_0 anymore, so the cleaner thread will terminate
  if(timer)
    {
    if(!CloseHandle(timer)) retval = error();
    timer = 0;
    }

  for(i = 0; i < MEM_LEN; i++)
    {
    if(memory[i].lock) if(!CloseHandle(memory[i].lock)) retval =
worst(retval, error());
    }

  if(threadhandle)
    {
  //we wait until the cleaner thread is done
    wait_for(threadhandle);
    CloseHandle(threadhandle);
    threadhandle = 0;
    }
  threadid = 0;

  for( i = 0; i < MEM_LEN; i++)
    {
  //and free every memory block which is still in the list
    for(j = 0; j < memory[i].count; j++)
      {
      if(!HeapFree(mainheap, H_F_F, memory[i].blocks[j].ptr))
                                          retval = worst(retval, error());
      }

    if(!HeapFree(mainheap, H_F_F, memory[i].blocks))
                                          retval = worst(retval, error());
    }

 //all global variables will be set to zero again, so one could restart
 //the memory manager savely
  RtlZeroMemory(&memory, sizeof(memory));
  RtlZeroMemory(&ms, sizeof(ms));
  mainheap = 0;

  return retval;
  }


c_dword        FC max_ram                      (void)
  {
  return ms.total_phys;
  }


c_dword        FC max_virtual                  (void)
  {
  return ms.total_virtual;
  }
```

```
c_dword       FC free_ram                   (void)
  {
  GlobalMemoryStatus(ms);
  return ms.avail_phys;
  }


c_dword       FC free_virtual               (void)
  {
  GlobalMemoryStatus(ms);
  return ms.avail_virtual;
  }


c_hresult     FC allocate                   (c_dword       asize,
                                             r_pointer     amem)
  {
  hresult retval = S_OK;

  if(asize > 0)
    {
    //if no fitting block can be found, we use the old fashioned HeapAlloc
    amem = find(asize);
    if(!amem)
      {
      p_dword p = RCAST(p_dword, HeapAlloc(mainheap, H_A_F,
                                           asize + sizeof(dword)));

      if(p)
        {
        p[0] = asize;
        amem = NORM_PTR(p);
        }
      else retval = error();

      }
    }
  else
    {
    retval = S_FALSE;
    amem   = NULL;
    }

  return retval;
  }


void          FC free                       (r_pointer     amem)
  {
  if(amem)
    {
    enqueue(ORIG_PTR(amem));
    amem = 0;
    }
  }
```

```
c_hresult     FC reallocate                    (c_dword        anewsize,
                                                r_pointer      amem)
  {
  hresult retval = S_OK;

  if(amem)
    {
    if(anewsize > 0)
      {
      p_dword p = ORIG_PTR(amem);
      dword   s = p[0];

      if(anewsize != s)
        {
        //we shrink a memory block only, when it's new size comes
        //shorter than one third of it's actual size
        if((anewsize * 3) < s) s = anewsize;

        //so if this happens, or the new size is even bigger than
        //the old,
        if(anewsize >= s)
          {
        //we either find a proper block in the list
          pointer d = find(anewsize);
          if(d)
            {
            RtlMoveMemory(d, &p[1], s);
        //and exchange it with the old
            enqueue(p);
            amem = d;
            }
          else
            {
        //or really need to reallocate
            p_dword e = RCAST(p_dword, HeapReAlloc(mainheap, H_RA_F,
                                       p, anewsize + sizeof(dword)));
            if(e)
              {
              e[0] = anewsize;
              amem = NORM_PTR(e);
              }
            else retval = error();
            }
          }
        }
      }
    else
      {
      free(amem);
      retval = S_OK;
      }
    }
  else
    {
    retval = allocate(anewsize, amem);
    }

  return retval;
  }
```

```
c_dword        FC size                         (c_pointer      amem)
  {
  if(amem) return ORIG_PTR(amem)[0];
  return 0;
  }
```

3.4 test.cpp

With test.cpp, a simple test program for the heap manager's performance is given. `THREAD_COUNT` defines how many threads should run simultaneously in the test, `TEST_TIMES` defines how often the single tests should be repeated and `TEST_LEN` says how many allocations, reallocations and freeing operations shall be done in one single test. The program uses a simple random number generator for every thread and also suspends the process in sleep cycles (a few minutes = `CALM_TIME`) between the tests to ensure that the tests cannot influence each other with e.g. down-clocking, disk operations due to paging.
Of course this test cannot represent the situation in a multi-thread application completely correctly, since memory operations occur not that consecutively, but it shows how performance *can* be increased by the heap manager.

```
namespace _malloc
  {
#include <malloc.h>
  }

#include "basics.h"
#include "HeapManager.h"
#include <stdio.h>


#define THREAD_COUNT  40
#define TEST_TIMES    1000
#define TEST_LEN      600
#define CALM_TIME     (10 * 60 * 1000)
#define CALC_TIME(a)  ( ((double)a) / ((double)10000000) )
dword   max_alloc;


#define MAX_RAND     (((hugeint)1 << (hugeint)61) - (hugeint)1)
#define RAND_MUL     (((hugeint)1 << (hugeint)40) - (hugeint)87)
#define RAND_ADD     (((hugeint)1 << (hugeint)53) - (hugeint)111)

//every thread gets it's own random number generator
//so we don't need to synchronize randomization
hugeint  randseed[THREAD_COUNT];

c_double     FC     random          (c_dword idx)
  {
  randseed[idx] = ((randseed[idx] * RAND_MUL) + RAND_ADD) % MAX_RAND;

  if(randseed[idx] < (hugeint)0) randseed[idx] = -randseed[idx];

  return ((double)randseed[idx]) / ((double)MAX_RAND);
  }


c_dword      FC     random_dword    (c_dword idx, c_dword amax)
  {
  return (dword)(random(idx) * (double)amax);
  }
```

```
//with that, we test the new memory manager
c_dword       SC test_thread_1(c_dword idx)
  {
  pointer ptrs[TEST_LEN];
  integer i, j;

  for(j = 0; j < TEST_TIMES; j++)
    {
    for(i = 0; i < TEST_LEN; i++) allocate(random_dword(idx, max_alloc),
                                           ptrs[i]);
    for(i = 0; i < TEST_LEN; i++) reallocate(random_dword(idx, max_alloc),
                                             ptrs[i]);
    for(i = 0; i < TEST_LEN; i++) free(ptrs[i]);
    }

  return 0;
  }


//to get comparable results, we also test HeapAlloc & Co. which our memory
//manager uses
c_dword       SC test_thread_2(c_dword idx)
  {
  pointer ptrs[TEST_LEN];
  integer i, j;

  for(j = 0; j < TEST_TIMES; j++)
    {
    for(i = 0; i < TEST_LEN; i++) ptrs[i] = HeapAlloc(GetProcessHeap(),0,
                                            random_dword(idx,max_alloc));
    for(i = 0; i < TEST_LEN; i++) ptrs[i] = HeapReAlloc(GetProcessHeap(),
                                            0, ptrs[i],
                                            random_dword(idx,max_alloc));
    for(i = 0; i < TEST_LEN; i++) HeapFree(GetProcessHeap(), 0, ptrs[i]);
    }
  return 0;
  }


//and also we test GlobalAlloc, which is only for compatibility still alive
//and should not be used anymore
//since it is a little bit slower than HeapAlloc, i think it is just a wrapper
//for HeapAlloc, at least under winxp which i have
c_dword       SC test_thread_3(c_dword idx)
  {
  pointer ptrs[TEST_LEN];
  integer i, j;

  for(j = 0; j < TEST_TIMES; j++)
    {
    for(i = 0; i < TEST_LEN; i++) ptrs[i] = GlobalAlloc(GMEM_FIXED,
                                            random_dword(idx,max_alloc));
    for(i = 0; i < TEST_LEN; i++) ptrs[i] = GlobalReAlloc(ptrs[i],
                                            random_dword(idx,max_alloc),
                                            GMEM_MOVEABLE);
    for(i = 0; i < TEST_LEN; i++) GlobalFree(ptrs[i]);
    }

  return 0;
  }
```

```
event mallocevent;

//finally, we compare with malloc and co.
//but we need to synchronize explicitely, else we crash
c_dword       SC test_thread_4(c_dword idx)
  {
  pointer ptrs[TEST_LEN];
  integer i, j;

  for(j = 0; j < TEST_TIMES; j++)
    {
    for(i = 0; i < TEST_LEN; i++)
      {
      WaitForSingleObject(mallocevent, INFINITE);
      ptrs[i] = _malloc::malloc(random_dword(idx,max_alloc));
      SetEvent(mallocevent);
      }

    for(i = 0; i < TEST_LEN; i++)
      {
      WaitForSingleObject(mallocevent, INFINITE);
      ptrs[i] = _malloc::realloc(ptrs[i], random_dword(idx,max_alloc));
      SetEvent(mallocevent);
      }

    for(i = 0; i < TEST_LEN; i++)
      {
      WaitForSingleObject(mallocevent, INFINITE);
      _malloc::free(ptrs[i]);
      SetEvent(mallocevent);
      }
    }

  return 0;
  }


//this function runs the multi-thread tests
//it returns the time the test needed on a 100 nanosecond scale
filetime test(c_pointer afunc)
  {
  thread   threads[THREAD_COUNT];
  dword    i, j;
  filetime s, e;

  GetSystemTimeAsFileTime(s);

  for(i = 0; i < THREAD_COUNT; i++)
     threads[i] = CreateThread(NULL, 0, afunc, (pointer)i, 0, j);

  WaitForMultipleObjects(THREAD_COUNT, (pc_handle)&threads,
                         true, INFINITE);

  GetSystemTimeAsFileTime(e);

  for(i = 0; i < THREAD_COUNT; i++) CloseHandle(threads[i]);

  return e - s;
  }
```

```
MAINA
  {
  filetime hm, ha, ga, ma;
  hm=0; ha=0; ga=0; ma=0;

  //to make sure our test don't exceeds the free memory
  //we calculate the maximum allocation size
  //the 3 * ... / 2 comes because the random number generator
  //generates an average of about 0.5, so we can savely go up to 0.66
  max_alloc = (3 * free_ram()) / (2 * THREAD_COUNT * TEST_LEN);

  //initialize the random seeds
  GetSystemTimeAsFileTime(hm);
  for(dword i = 0; i < THREAD_COUNT; i++) randseed[i] = hm;

  //our memory manager needs to be initialized and closed
  init_memory_management();
  //to let the system "calm down", meaning to let it end all pending
  //operations like file stuff, pageing, network or whatever,
  //we let our program sleep a little bit
  //afterwards tests are surely not influenced by any stuff we did before
  Sleep(CALM_TIME);
  //run the 1st test
  hm = test(&test_thread_1);
  close_memory_management();
  printf("\nHeapManager                    : %f s.\n", CALC_TIME(hm));

  //again we sleep, this time we asure that the system can "cool down",
  //because we stressed it a little bit with the last test
  //so any cpu down clocking which could have been performed will
  //not influence the next test
  //also any pageing or such we could have caused will be finished
  Sleep(CALM_TIME);
  ha = test(&test_thread_2);
  printf("\nHeapAlloc                      : %f s.\n", CALC_TIME(ha));

  Sleep(CALM_TIME);
  ga = test(&test_thread_3);
  printf("\nGlobalAlloc                    : %f s.\n", CALC_TIME(ga));

  //after sleeping we test malloc, but need to synchronize explicitely
  mallocevent = CreateEventW(NULL, false, true, NULL);
  Sleep(CALM_TIME);
  ma = test(&test_thread_4);
  printf("\nmalloc                         : %f s.\n", CALC_TIME(ma));
  CloseHandle(mallocevent);

  //to make results easily comparable
  printf("\nHeapManager vs. HeapAlloc   : 1:%f\n",
              (double)((hugeint)ha)/(double)((hugeint)hm));
  printf("\nHeapManager vs. GlobalAlloc: 1:%f\n",
              (double)((hugeint)ga)/(double)((hugeint)hm));
  printf("\nHeapManager vs. malloc      : 1:%f\n",
              (double)((hugeint)ma)/(double)((hugeint)hm));


  char xxxxxxx = 0;
  xxxxxxx = (char)scanf("%c", &xxxxxxx);
  return 0;
  }
```