condat®

Enhancing Windows Shared Memory for VCMS (The SHM-NT Gadget)

“… ‘cause windows *memory-mapped file* solution is a lame

duck!”

Introduction

- Problems in VCMS with using Memory-Mapped Files
- Fundamentals of the new implementation
- New problems to resolve
- Source glimpses
- Interface
- Discussion

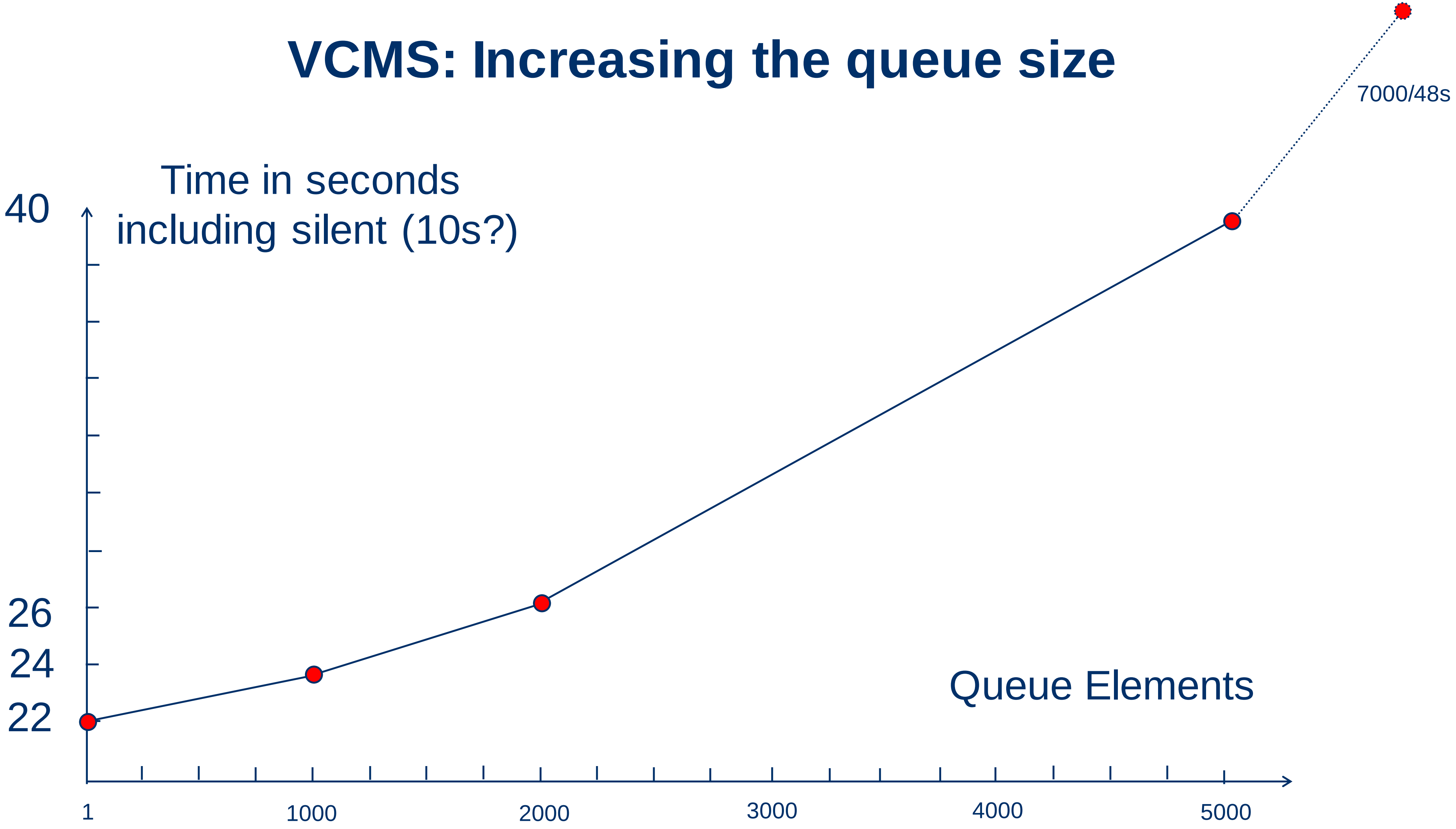
Problems with using Memory-Mapped Files (1)

- **Linking on fixed Address required for some DLL’s**
  - ⇨ pointer issues on different address-ranges requires static linkage
    - ◆cms.dll: /base:0x20000000 /fixed
    - ◆frame.dll: /base:0x60100000 /fixed
- **Performance Issues**
  - ⇨ some G23Net test cases failed with time out
  - ⇨ stunning result after foolish fiddling with parameters:
    - ◆ ***The bigger the queue***
      - ***the slower the transfer?!?***

Problems with using Memory-Mapped Files (2)

Confidential

VCMS: Increasing the queue size



Problems with using Memory-Mapped Files (3)

- **Why?**
- **Assumption:**
  - ⇨ the bigger size of the queue seems to provoke a higher system utilization
- **Quick guesses:**
  - ⇨ caused by an additional abstraction layer of file mapped shared memory
    - ◆ which *“covers”* the shared memory system
  - ⇨ a bad scheduling provokes dispatching of idle processes
- **Target solution**
  - ⇨ elimination of the additional abstraction layer

Problems with using Memory-Mapped Files (4)

- **Additional Abstraction-Layer of Memory-Mapped Files**
  - ⇨ CreateFile()
  - ⇨ CreateFileMapping()
  - ⇨ MapViewOfFile()
- **uses undocumented shared memory internal functions**
  - ⇨ ZwCreateSection()
  - ⇨ ZwMapViewOfSection()

Fundamentals of the new implementation

- **Frank Reglin’s sample application:**
  - ⇨ focuses the internal management of shared memory
    - ◆ named sections
    - ◆ linked list of sections
    - ◆ managing section contents via alloc/free
  - ⇨ base usage of Zw*() functions
- **Some undocumented Windows NT/2000 Zw*() functions:**
  - ⇨ ZwCreateSection()
  - ⇨ ZwOpenSection()
  - ⇨ ZwMapViewOfSection(), ZwUnmapViewOfSection()
  - ⇨ missing:
    - ◆ ZwDeleteSection()
    - ◆ ZwCloseSection()

New problems to resolve (1)

- **ZwMapViewOfSection() doesn’t guarantee a unique mapped location for all views of the same memory section**
  - ⇨ a main goal of our implementation, enhancing the Windows shared mem
- **but how to achieve?**
  - ⇨ ZwMapViewOfSection(hdl, …, &addr, …) with addr == 0 means automatic view map placement
  - ⇨ addr returns the resulting location of this premier placement
  - ⇨ which mustn’t change for all further ZwMapViewOfSection() calls with the same handle
  - ⇨ and as to be propagated to all clients, hence
- **Win 2000 rejects automatic view map placement**
  - ⇨ an incremental, aligned placement has to be applied instead
  - ⇨ till mapping of a premier view placement succeeds

New problems to resolve (2)

- OK, an initial map view placement of a given section may succeed
- How to propagate the location to all clients of interest?
  - ⇨ Message Passing?
  - ⇨ temporary File?
  - ⇨ other IPC mechanism?
- Why not use the new shared memory gadget?
  - ⇨ constituting a ***pool list*** containing section addresses inside a specific, qualified shared memory section?

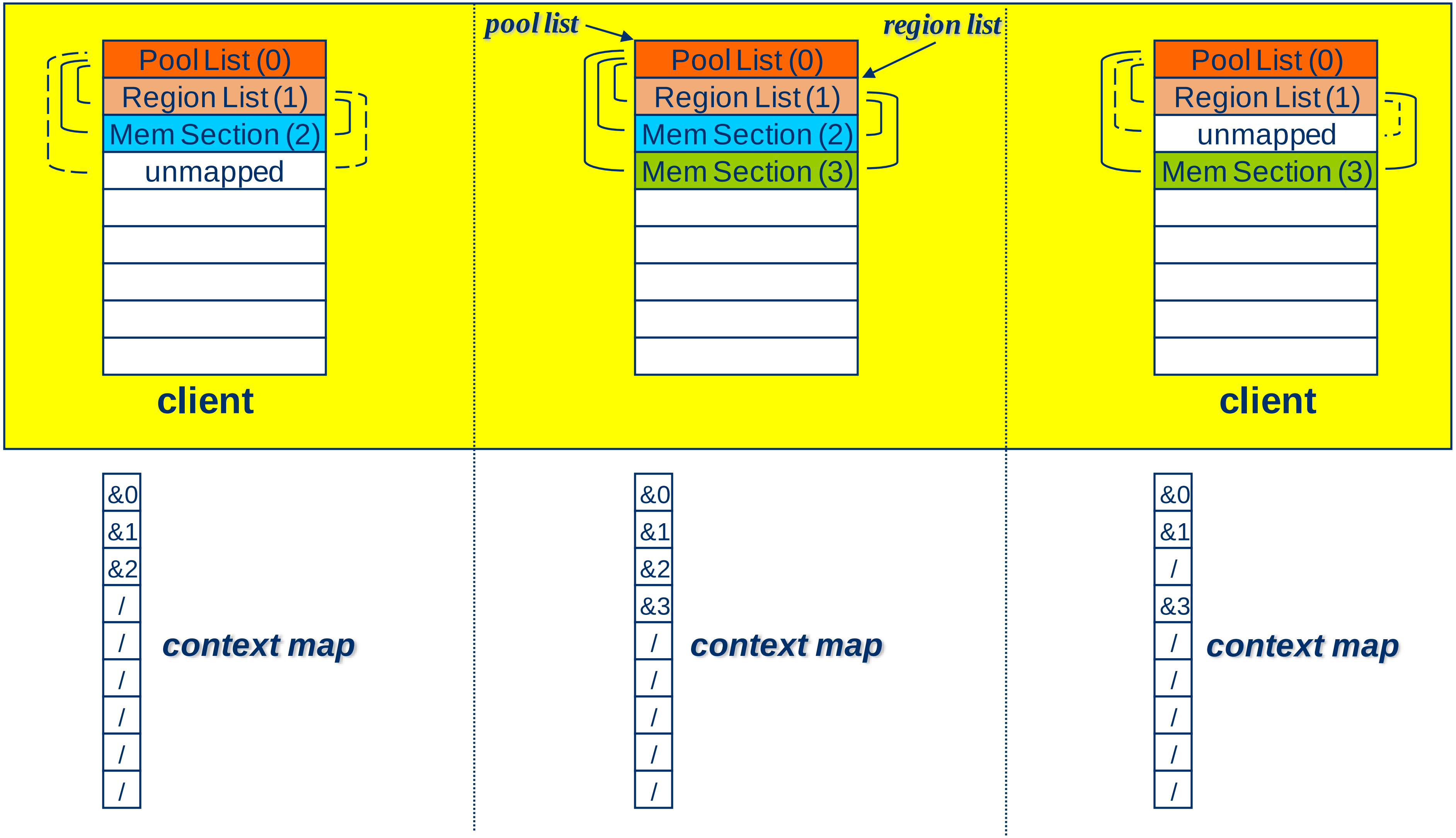
New problems to resolve (3)

- **There is more implementation-specific shared knowledge**
  - ⇨ uncritical global scalar data types
    - ◆ and pointers referencing unshared data
    - ◆ which can be grouped and shared in a DLL by use of a link command
  - ⇨ and critical: pointers referencing a shared object
    - ◆ e.g. fr’s ***region list*** constituting a shared memory list
    - ◆ cause DLL’s also suffers the windows map view location weakness
  - ⇨ again: Why not use the new shared memory implementation?
    - ◆ constituting the ***region list*** inside a specific, qualified shared memory section?
- **And local knowledge reflecting parts of global knowledge**
  - ⇨ a client needs to know which sections are already mapped

New problems to resolve (4)

- **But: How to use the new shared memory gadget**
  - ⇨ to implement itself?
  - ⇨ isn't that a hen/egg problem?
- **No, it's just a matter of proper initializing and embedding**
  - ⇨ both special sections (***pool list***, ***region list***) are known to the system
  - ⇨ *those internal* sections receive a special treatment whilst system start-up
- **the *pool list* is just an array**
  - ⇨ containing section names, addresses and sizes
- **the *region list* is a double linked list**
  - ⇨ but its anchor is located outside of it's shared memory section
  - ⇨ constituted by a DLL global variable

The Big Picture



pool list

region list

Pool List (0)
Region List (1)
Mem Section (2)
unmapped

client

&0
&1
&2
/
/
/
/
/
/

context map

Pool List (0)
Region List (1)
Mem Section (2)
Mem Section (3)

&0
&1
&2
&3
/
/
/
/
/

context map

Pool List (0)
Region List (1)
unmapped
Mem Section (3)

client

&0
&1
/
&3
/
/
/
/
/

context map

Source glimpses (create & map a Section)

```
create_attrib_object(name, &obj);
ZwCreateSection( &hdl, ..., &obj, ...);
ZwOpenSection( &hdl, ..., &obj  );
```

Case 1) get_from_pool(name, &tmpaddr) == FALSE:

```
if (osvi.dwMajorVersion == WIN2000 ) {
  mappedAddr = shm_offset();
                    /* already stored sections */
} else if (osvi.dwMajorVersion == WINNT) {
  mappedAddr = 0L; /* use the first free area. */
}

rc = ZwMapViewOfSection( hdl, (HANDLE)-1,
      &mappedAddr, ...);
if (rc != STATUS_SUCCESS) {
  if (mappedAddr == 0)  {
    mappedAddr = SHAREDEND;
  }
  /* incremental, aligned placement */
 for( ; mappedAddr != SHAREDBEGIN;
                mappedAddr -= SHAREDSTEP ) {
   rc = ZwMapViewOfSection( hdl, (HANDLE)-1,
     mappedAddr, ... );
   if( rc == STATUS_SUCCESS )
     break;
  }
}
put_to_pool(name, mappedAddr, rsize)
```

Case 2) get_from_pool(name, &tmpaddr) == TRUE:

```
mappedAddr = (char*) tmpaddr;
ZwMapViewOfSection( hdl, (HANDLE)-1,
      &mappedAddr, ...);
```

Both cases:

```
notice_mapping_in_context(mappedAddr);
```

Source glimpses (creation of a Shared Heap)

```
if (poolAddr == 0) {
  init_shared_address_pool(); /* give access to
        the 2 internal regions */
}

shm_map_all();

reg = regionlist;
while( reg ) {
  pr = reg->pregion;
  if( strcmp(pr->name, name) == 0 ) {
    *phdl = (unsigned long)pr;
    return SHM_EXISTS;
  }
  reg = reg->next;
}
```

```
shm_section( name, size, &addr);

... init internal heap managment data ...

reg = shm_alloc( poolListAddr,
        sizeof(USEDREGION));

reg->pregion = addr;
reg->next = regionlist;
regionlist = reg;
```

Source glimpses (init of internal sections)

Case 1) regionlist == 0;

```
ret0 = shm_section((char*)REGIONPOOLNAME,
      poolSize0, &poolAddr );

/* local var poolAdr is != NULL from now on,
      will trigger action of the other pool funcs ...
      */

if ((ret0 == SHM_OK) || (ret0 == SHM_EXISTS)) {
  region0 = poolAddr;
}

ret1 = shm_heap((char*)REGIONLISTNAME,
      poolSize1, &poolListAddr, TRUE);

if ((ret1 == SHM_OK) || (ret1 == SHM_EXISTS)) {
  if (ret0 == SHM_OK)
  {
    region1 = poolListAddr;
    put_to_pool(REGIONPOOLNAME, poolAddr,
      poolSize0);
    while((spe++) <= lastSpe) {
      spe->name[0] = 0;
      spe->addr = 0;
      spe->size = 0;
    }
    reg = shm_alloc( poolListAddr,
      sizeof(USEDREGION));
    reg->next = regionlist;
    reg->pregion =
      (REGIONGLOBALS*)poolListAddr;
    regionlist = reg;
  }
}
```

Case 2) regionlist != 0;

```
shm_map((char*)REGIONPOOLNAME,
      poolSize0, &region0);
poolAddr = region0;

shm_map((char*)REGIONLISTNAME, poolSize1,
      &region1);
poolListAddr = region1;
```

Interface (1)

- **int shm_section(char * name, unsigned long int rsize, unsigned long int * phdl);**
  - ⇨ creates a “raw” (to be managed by user) *shared memory section*
  - ⇨ no malloc/free available
- **int shm_heap(char * name, unsigned long int rsize, unsigned long int * phdl, BOOL forceInit);**
  - ⇨ creates a *shared memory heap*
    - ◆ providing classic malloc/free
    - ◆ based on shm_section()
- **int shm_delete_section(unsigned long int hdl);**
  - ⇨ deletes a shared memory section
  - ⇨ or a shared heap

Interface (2)

- **void * shm_alloc(unsigned long int hdl, unsigned long int size);**
  - ⇨ allocates a chunk in the denoted shared heap
- **int shm_free(void * addr);**
  - ⇨ releases a chunk in the denoted shared heap
- **int shm_exit();**
  - ⇨ unmaps all sections from a client
  - ⇨ does not delete any section
    - ◆ irrespective the internal sections
      - which are deleted, if they are the solely remainder

Interface (3)

- **int shm_map(char * name, unsigned long int rsize, unsigned long int * paddr);**
  - map a single, specific shared memory section/heap.
- **void shm_map_all();**
  - map ALL remote created shared memory sections
  - into current address space
- **long int shm_map_by_exeption(EXCEPTION_POINTERS* EP);**
  - map ALL remote created shared memory sections by "*trap on use*"

```
 int q_read(
__try{
  [...main q_read code...]
  }
  __except (shm_map_by_exeption(GetExceptionInformation())) {
  }
)
```

Interface (4)

- **void shm_list_pools();**
  - ⇨ print information about all shared memory address pools (***pool list***):
    - name
    - address
    - size
    - range

- **void shm_list_heap(FILE * outf, char * name);**
  - ⇨ service/debug. List internal management data of a shared memory heap.
  - ⇨ used in Program *shmList.exe*, not really necessary for implementation.

Why dynamic-link-libraries for sharing?

- **We’ve handled shared dynamic data. What’s about linking?**
  - ⇨ can we get rid of that clumsy /base:0x60100000 /fixed linkage now?
  - ⇨ ***YES!*** But hasn’t **SHM_NT** have to handle the shared static data, too? By moving it all into shared memory sections?
    - ◆ ***NO!*** We can safely ignore this type of FRAME data:
      - if it is located and referenced in the *stack* only (single process)
      - if it is ***non-***win32 code, like all partition-memory related data *(by now! Partitions may come to win32 later on)*
      - if the data (or any sub-data, if structured) isn’t remembered by it’s location (& address operator and resulting pointer)
- **Why providing a DLL-solution only (no shm_nt.lib)?**
  - ⇨ **SHM_NT** uses VCMS semaphores, based on shared data. Easily achieved by constituting a dll. Suggestion for a coming VCMS-Release: dynamic semaphore creation based on a SHM_NT

Results

- **We earn a better performance now**
  - ⇨ which is at least, say, 30 % faster than the best of previous
  - ⇨ rather independent from queue buffer element size
- **Why isn’t performance gain higher on larger queues?**
  - ⇨ good question!
  - ⇨ but why should it?
    - ◆ de-coupling is properly done
      - ● by eliminating the Windows 2nd abstraction level
    - ◆ No further scheduling/dispatching problems
    - ◆ we have only one Processor
- **Discussion, anybody?**

How to access and use the SHM-NT Gadget

- Sources:
  - ⇨ **\gpf\shm_nt\…**
  - ⇨ **\gpf\vcms-nt\…** (example of usage)
- Includes:
  - ⇨ **\gpf\shm_nt\inc\shm_nt.h**
- DLL:
  - ⇨ **\gpf\shm_nt\lib\shm_nt.dll**
- Lables (preliminary):
  - ⇨ **SHM_NT_FLOAT**
  - ⇨ **VCMS_FLOAT**

Further reading

- Jeffrey Richter, MS-Windows für Experten
  - ⇨ An introduction into Windows System Programming
- Gary Nebbett, WINDOWS NT/2000 Native API Reference
  - ⇨ Win NT/2K undocumented system calls
- Randy Kath, Managing Virtual Memory in Win32
  - ⇨ http://msdn.microsoft.com/library/en-us/dngenlib/html/msdn_virtmm.asp
- Randy Kath, Managing Memory-Mapped Files in Win32
  - ⇨ http://msdn.microsoft.com/library/en-us/dngenlib/html/msdn_manamemo.asp

condat®

End

(Enough for today)