Win32 API Emulation on UNIX for Software DSM

Sven M. Paas, Thomas Bemmerl, Karsten Scholtyssik
Lehrstuhl für Betriebssysteme, RWTH Aachen
Kopernikusstr. 16, D-52056 Aachen, Germany
Email: `contact@lfbs.rwth-aachen.de`
WWW: `http://www.lfbs.rwth-aachen.de/`

Abstract. *This paper presents a new Win32 emulation layer called nt2unix. It supports source code compatible Win32 console applications on UNIX. We focus on the emulation of specific Win32 features used for system programming like exception handling, virtual memory management, NT thread synchronization and the WinSock API. As a case study, we ported the all-software DSM system SVMlib - consisting of about 15.000 lines of C++ code written natively for the Win32 API - to Sun Solaris with absolutely no source code changes.*

1 Introduction

While there exist numerous toolkits designed for porting software from UNIX to Windows NT on source code level [14], little effort has be conducted to provide a toolkit to port Windows NT applications, especially those using the Win32 API directly, to UNIX. With the emerging importance of Windows NT [8], more and more applications are newly developed for the Win32 API, so a migration path to support the various UNIX flavors even for low level applications is highly desirable. In this paper, we propose a library based approach to achieve source code level compatibility for a specific subset of the Win32 API under the UNIX operating system.

2 The nt2unix Emulation Layer

In this section, we introduce the functionality of nt2unix and its strategy to implement a relevant subset of the Win32 API on Solaris [12], a popular UNIX System V implementation. Since a complete implementation of the Win32 API under UNIX is not practicable, we had to decide which features to support. As our focus lies on systems programming, we chose the following function groups to form a reasonable subset:

- *NT Thread Management*. This group includes functions for creating, destroying, suspending and resuming preemptive threads. It also includes functions to synchronize concurrent threads and TLS (thread local storage) functions.
- *Virtual Memory Management*. This group includes the interface to the virtual memory (VM) manager as well as functions for memory mapped I/O.
- *Error Handling*. Win32 supports user level handlers to catch special exceptions as well as special error handling routines. These functions form another group to be supported.
- *Networking*. This group concerns networking, which includes the complete WinSock API.

Naturally, an emulation layer firstly has to support the basic data types found in the Win32

API. nt2unix supports most specific simple Win32 data types, like DWORD, BOOL, BYTE and so on. The much more interesting problems arise from the implementation of certain functions. Some specific problems we encountered are presented in the next sections.

2.1 NT Multithreading and Synchronization

In order to support NT multithreading, nt2unix must keep track of thread associated data normally the NT kernel stores. This data includes:

- The state of a thread (running, suspended or terminated);
- The *suspend counter* of a thread (a concept unknown in Solaris);
- The exit code of the thread.

nt2unix uses the STL type `map` to store the above information for each thread. The entries in the map are indexed by the NT handle of the thread. Accesses to this map a protected by a special lock object:

```
typedef map<HANDLE, ThreadInfo, less<HANDLE> > ThreadInfoMap;
static ThreadInfoMap ThreadInfos;
static CriticalSection ThreadInfoLock;
```

Severe problems occur in order to support the Win32 functions **SuspendThread**() and **ResumeThread**(). At first glance, it seems obvious that these two functions can easily be emulated by the Solaris functions **thr_suspend**(3T) and **thr_resume**(3T). However, this is not the case, since there is a lost signal problem to be avoided when a thread suspends.

To understand this, we have a deeper look at our implementation of **SuspendThread**(). When this function is called, the lock protecting the thread data is acquired. Afterwards, the suspend counter of the thread is incremented, if possible. If the old suspend count is zero, two cases may occur: the thread may suspend itself or another thread. If the first case is true, the lock is released before actually calling **thr_suspend**() to avoid deadlock. In the second case, a lost signal problem must be avoided, since under Solaris, resuming threads doesnot work in advance, that is, resume actions are not queued if the target thread is not yet suspended at all. Our solution to this problem is to let the **ResumeThread**() implementation poll until the thread which has to be resumed has indicated its new state by setting a special flag, threadHasBeenResumed. So the code for **SuspendThread**() is like the following:

```
DWORD SuspendThread(HANDLE hThread) {
    BOOL same = FALSE; // this flag indicates whether a thread suspends itself.
    // If same == TRUE, we must avoid a „lost signal“ problem, see below.
    ThreadInfoLock.enter();
    ThreadInfoMap::iterator thisThreadInfo = ThreadInfos.find(hThread);
    if (thisThreadInfo != ThreadInfos.end()) { // found it.
        DWORD oldSuspendCount = (*thisThreadInfo).second.suspendCount;
        if (oldSuspendCount < MAXIMUM_SUSPEND_COUNT)
            (*thisThreadInfo).second.suspendCount++;
        if (oldSuspendCount < 1) {
            (*thisThreadInfo).second.state = THREAD_SUSPENDED;
            if (same = (thr_self() == (thread_t)hThread)) {
                // if the thread suspends itself, we must release the lock.
                (*thisThreadInfo).second.threadHasBeenResumed = FALSE;
                ThreadInfoLock.leave();
            }
            // DANGER!!! If at this point, another thread is scheduled in ResumeThread(), the
            // resume „signal“ may get lost. To avoid this, ResumeThread() polls until the thread is
            // really resumed, i.e. until threadHasBeenResumed == TRUE.
            if (thr_suspend((thread_t)hThread)) {
                perror(„thr_suspend()“); return 0xFFFFFFFF;
            }
            (*thisThreadInfo).second.threadHasBeenResumed = TRUE;
            if (!same)
                ThreadInfoLock.leave();
        } else  // thread is already sleeping
            ThreadInfoLock.leave();
        return oldSuspendCount;
    }
    // Thread not found.
    ThreadInfoLock.leave();
    return 0xFFFFFFFF;
}
```

The corresponding **ResumeThread**() code is as follows:

```
DWORD ResumeThread(HANDLE hThread) {
    ThreadInfoLock.enter();
    ThreadInfoMap::iterator thisThreadInfo = ThreadInfos.find(hThread);
    if (thisThreadInfo != ThreadInfos.end()) { // found it.
        DWORD oldSuspendCount = (*thisThreadInfo).second.suspendCount;
        if (oldSuspendCount > 0) {
            (*thisThreadInfo).second.suspendCount--;
            if (oldSuspendCount < 2) {
                // oldSuspendCount == 1 -> new value is 0 -> really resume thread
                (*thisThreadInfo).second.state = THREAD_RUNNING;
                do { // Loop until the target thread is really resumed.
                    if (thr_continue((thread_t)hThread)) {
                        ThreadInfoLock.leave();
                        return 0xFFFFFFFF;
                    }
                    // Give up the CPU so that the resumed thread has a chance to
                    // update the associated threadHasBeenResumed flag.
                    thr_yield();
                } while (!(*thisThreadInfo).second.threadHasBeenResumed);
            }
        }
        ThreadInfoLock.leave();
        return oldSuspendCount;
    } // thread not found.
    ThreadInfoLock.leave();
    return 0xFFFFFFFF;
}
```

2.2 Virtual Memory Management

Win32 supports an interface to the VM system, especially to protect and map virtual memory pages. Like for threads, nt2unix has to keep track of data for each file mapping in the system. nt2unix stores the following information for each mapping:

```
struct FileMapping {
    LPVOID lpBaseAddress;                   // base address of mapping
    DWORD dwNumberOfBytesToMap;             // mapping size
    HANDLE hFileMappingObject;              // file handle
    char FileName[MAX_PATH];                // file name
    DWORD refcnt;                           // number of references to the mapping
};
static vector<FileMapping> FileMappings;
```

Using a STL-style `vector` of mappings, NT mapping is easily achieved by using mmap():

```
WINBASEAPI LPVOID WINAPI MapViewOfFileEx(
                HANDLE hFileMappingObject, DWORD dwDesiredAccess,
                DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow,
                DWORD dwNumberOfBytesToMap, LPVOID lpBaseAddress ) {
    int prot = 0, flags = 0; LPVOID ret;
    if (dwFileOffsetHigh > 0)
        DBG(„MapViewOfFileEx(): ignoring dwFileOffsetHigh“);
    // Filter the protection bits and mapping flags ...
    prot =dwDesiredAccess & FILE_MAP_ALL_ACCESS;
    flags = dwDesiredAccess & FILE_MAP_COPY ? MAP_PRIVATE : MAP_SHARED;
    if (lpBaseAddress)
        flags |= MAP_FIXED;
    // Search and update the mapping in the vector.
    vector<FileMapping>::iterator i=FileMappings.begin();
    while(i != FileMappings.end() &&
                i->hFileMappingObject != hFileMappingObject)
        i++;
    if (i != FileMappings.end()) {
        if (dwNumberOfBytesToMap)
            i->dwNumberOfBytesToMap = dwNumberOfBytesToMap;
    } else
        return 0;
    if ((ret = (LPVOID)mmap((caddr_t)lpBaseAddress,
                (size_t)i->dwNumberOfBytesToMap, prot, flags,
                (int)hFileMappingObject, (off_t)dwFileOffsetLow))
                == (LPVOID)MAP_FAILED) {
```

```
        return 0;
    }
    if (mprotect((caddr_t)ret, (size_t)i->dwNumberOfBytesToMap, prot) == -1)
            perror(„mprotect()“);
    return ret;
}
```

2.3 NT Exception Handling

Windows NT provides two means of delivering exceptions to user level processes:

- by embracing the code with a __try{} ... __except(){} block.
- by installing an exception handler calling **SetUnhandledExceptionFilter**().

The second method is supported by nt2unix. Exceptions are mapped to semantically more or less equivalent UNIX-style signals, like denoted in the following table. Note that not all exception codes of Windows NT have meaningful counterparts in a UNIX environment.

Type of exception	**NT Exception Code**	**UNIX Signal**
Access Violation	EXCEPTION_ACCESS_VIOLATION	SIGSEGV
Floating Point Exc.	EXCEPTION_FLT_INVALID_OPERATION	SIGFPE
Illegal Instruction	EXCEPTION_ILLEGAL_INSTRUCTION	SIGILL
Bus Error	EXCEPTION_IN_PAGE_ERROR	SIGBUS
Trace Trap	EXCEPTION_SINGLE_STEP	SIGTRAP

2.4 Summary

The following table shows a summary of all functions implemented within nt2unix.

	Win32 Functions emulated	**Emulation is based on**
NT Multithreading	CreateThread()	thr_create()
	GetCurrentThread()	thr_self()
	GetCurrentThreadId()	thr_self()
	ExitThread()	thr_exit()
	TerminateThread()	thr_kill()
	GetExitCodeThread()	STL
	SuspendThread()	thr_self(), thr_suspend()
	ResumeThread()	thr_yield(), thr_resume()
	Sleep()	thr_yield(), thr_suspend(), poll()
NT Thread Synchronization	InitializeCriticalSection()	mutex_init()
	DeleteCriticalSection()	mutex_destroy()
	EnterCriticalSection()	mutex_lock()
	LeaveCriticalSection()	mutex_unlock()

	Win32 Functions emulated	**Emulation is based on**

Thread Local Storage (TLS)	TlsAlloc()
	TlsGetValue()
	TlsSetValue()
	TlsFree()
		thr_keycreate()
		thr_getspecific()
		thr_setspecific()
		pthread_key_delete()

NT Object Handles	CloseHandle()
	DuplicateHandle()
	WaitForSingleObject()
		close()
		dup(), dup2()
		thr_join()

Process Functions	GetCurrentProcess()
	GetCurrentProcessId()
	ExitProcess()
		getpid()
		getpid()
		exit()

VM Management	VirtualAlloc()
	VirtualFree()
	VirtualProtect()
	VirtualLock()
	VirtualUnlock()
		mmap(), valloc(), mprotect()
		mprotect(), free()
		mprotect(), memcntl()
		mlock()
		munlock()

Memory Mapped I/O	MapViewOfFile()
	MapViewOfFileEx()
	UnmapViewOfFile()
	CreateFileMapping()
		mmap()
		mmap()
		munmap()
		STL

Error Handling	WSAGetLastError()
	GetLastError()
	SetLastError()
	WSASetLastError()
		errno
		errno
		errno
		errno

WinSock API	WSAStartup()
	WSACleanup()
	closesocket()
	ioctlsocket()
	all BSD-style functions!
		-
		-
		close()
		ioctl()
		socket(5) family

Exception Handling	SetUnhandledExceptionFilter()
	GetExceptionInformation()
	UnhandledExceptionFilter()
		sigaction()

Miscellaneous	GetSystemInfo()
	GetComputerName()
	QueryPerformanceFrequency()
	QueryPerformanceCounter()
		sysinfo()
		gethostname()
		-
		gettimeofday()

3 A Case Study: SVMlib

3.1 Overview

SVMlib [9, 13] (*Shared Virtual Memory Library*) is an all-software, page based, user level shared virtual memory subsystem for clusters of Windows NT workstations. It is one of the first [6] [10] SVM systems for this operating system. The source code of SVMlib consists of about 15.000 lines of C++ code written natively for the Win32 API. The library has been designed to benefit from several Windows NT features like preemptive multithreading and support for SMP machines. Unlike most software DSM systems, SVMlib itself is truly multithreaded. It also allows to create several preemptive user threads to speed up the computation on SMP nodes in the cluster. Currently the library uses TCP/IP sockets for communication purposes but it will also support efficient message passing using Dolphins implementation of SCI.

SVMlib provides a C/C++ API that allows the user to create and destroy regions of virtual shared memory that can be accessed fully transparently. Also different synchronization primitives like barriers and mutexes are part of the API. To keep track of accesses to the shared regions, SVMlib handles page faults within the regions via structured exception handling provided by the C++ run time system of Windows NT.

At the current stage, two different memory consistency models are supported by three different consistency protocols. The first consistency model offers the widely used though fairly inefficient *sequential consistency* [7] model. This model is supported by single writer as well as multiple writer protocols. Secondly, the distributed lock based *scope consistency* [5] is implemented.

Our main goal in this project is to examine the impact of efficient distributed synchronization protocols on the performance of a SVM system.

3.2 Design of SVMlib

When designing a SVM system, several design choices have to be made. When we started this project our primary goal was to develop a highly flexible and extendable research instrument. We therefore decided to build SVMlib as a set of independent modules where each can be exchanged without influencing the other modules.

Another important choice was the platform to build SVMlib on. As Windows NT is a modern operating system with some interesting features like true preemptive kernel threads, SMP support and a rich API we decided to use workstations running Windows NT as primary platform. Figure 3.1 shows the overall design of SVMlib. On the top level four modules are used.

The first is the *memory manager* that handles the creation and destruction of shared memory regions, catches page faults and implements the memory dependent part of the user interface. The memory manager manages a set of regions where each region can use a different consistency model and coherence protocol.

The second part is the *lock manager* that provides an interface that allows to create and destroy primitives for distributed process synchronization - mutexes as well as global barriers and semaphores.

For internode communication purposes the *communicator* is used. The user will never directly use this module. It is for internal purposes only. The communicator provides a simple interface containing a barrier, a broadcast algorithm and the possibility to send messages to each other node. This module has been designed to be active itself. To take advantage of the SMP support of Windows NT the communicator uses threads to handle incoming messages.

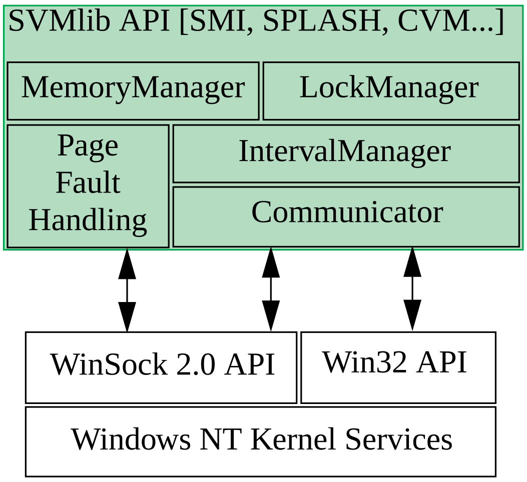

Figure 3.1: SVMlib components.

The last main module is the *interval manager* that allows to implement weak consistency models like lazy release consistency or the currently used scope consistency. The user will never have to access this module directly. It is used as a bridge between the memory and the lock manager when weak consistency models are used. This is needed because both locks and memory pages handle a part of the weak consistency model.

SVMlib provides several API personalities to the application programmer. First of all, a native C and C++ API is provided. For compatibility to other SVM systems and existing shared memory implementations, other interfaces to shared memory programming are supported. Currently, these interfaces include the *Shared Memory Interface* (SMI) [3], the macro interface of *Stanford Parallel Applications for Shared Memory* (SPLASH) [15] and the *Coherent Virtual Machine* (CVM) [11]. Other interfaces are planned to be supported in the future.

3.3 Performance Impact of the Emulation

Using nt2unix, we ported the source code of SVMlib to Sun Solaris 2.5.1 with *absolutely* no source code changes. This was very surprising, since, at first glance, a DSM implementation naturally is very system dependent. To show the impact of the Win32 emulation, we give usual metrics characterzing the performance of the library:

- *Page Fault Detection Time.* This value includes the mean time from the occurrence of a processor page fault on a protected page to the entrance of the handling routine. That is, this time includes all operating system overhead to deliver a page fault exception to user code. Note that there seems to be no difference between the NT Server and NT Workstation

	SuperSPARC, 50 MHz	**Pentium, 133 MHz**	**Pentium Pro, 200 MHz**
Windows NT 4.0 Server / Workstation	-	28 µs	19 µs
Solaris 2.5.1	135 µs	92 µs	48 µs

version with respect to exception handling. We compared these values with user level page fault detection under Solaris 2.5.1 for Intel and SPARC, respectively. Under UNIX, the memory exception handling mechanism of Windows NT is emulated by catching the `SIGSEGV` signal.

- *Page Fault Time*. This value includes the mean time to handle one page fault. This time excludes the page fault detection time mentioned above. It includes the overhead due to the coherence protocol and communication subsystem. In the current implementation, the times measured are mainly influenced by the high TCP/IP latency. The measurements were made using the FFT application of the set of CVM examples. This application implements a Fast Fourier Transformation on a 64 x 64 x 16 array. The coherence protocol used is a multiple reader / single writer protocol implementing sequential consistency. We compared three configurations running FFT: (1) *CVM on Solaris*: the CVM system running on Solaris 2.5.1, Sun SS-20, Ethernet; (2) *SVMlib on nt2unix*: the Solaris version of SVMlib, running on the same platform as (1), but with nt2unix emulation layer; (3) *SVMlib on Win32*: the native Win32 version of SVMlib, running on Windows NT 4.0, Intel Pentium-133, FastEthernet. Naturally, the Win32 time values mainly reflect the improved network performance of FastEthernet.

#Nodes	**Read / Write / Average Fault Time [ms] (CVM on Solaris)**	**Read / Write / Average Fault Time [ms] (SVMlib on nt2unix)**	**Read / Write / Average Fault Time [ms] (SVMlib on Win32)**
2	11.3 / 0.8 / 4.4	4.5 / 1.3 / 2.2	3.4 / 1.1 / 1.8
3	12.0 / 0.8 / 5.8	4.6 / 1.8 / 2.7	3.4 / 1.4 / 2.3
4	16.7 / 0.9 / 7.1	4.9 / 1.8 / 3.1	4.0 / 1.5 / 2.4

4 Summary and Conclusion

In this paper, we introduced nt2unix, a library providing an important subset of the Win32 API on UNIX based systems. The library makes it possible to port Win32 console applications to UNIX with much less effort. As a case study, we ported a complex DSM system with no source code changes at all from Windows NT to Solaris. We found that the performance impact of the emulation is not too high. The complete source code of the nt2unix library is available on request, please e-mail to `contact@lfbs.rwth-aachen.de`.

References

[1] Berrendorf, R.; Gerndt, M.; Mairandres, M.; Zeisset, S.: *A Programming Environment for Shared Virtual Memory on the Intel Paragon Supercomputer*, ISUG Conference, Albuquerque, 1995

[2] Dolphin Interconnect Solutions: *PCI-SCI Cluster Adapter Specification.* Jan. 1996.

[3] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *A Programming Interface for NUMA Shared-Memory Clusters.* Proc. High Perf. Comp. and Networking (HPCN), pp. 698-707, LNCS 1225, Springer, 1997.

[4] IEEE: *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI).* 1992.

[5] Iftode, L.; Singh, J. P.; Li, K.: *Scope Consistency: A Bridge between Release Consistency and Entry Consistency.* In Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA‘96), June 1996

[6] Itzkovitz, A., Schuster, A., Shalev, L.: *Millipede: a User-Level NT-Based Distributed Shared Memory System with Thread Migration and Dynamic Run-Time Optimization of Memory References*, Proc. of the USENIX Windows NT Workshop, Seattle, 1997

[7] Lamport, L.: *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28(9), pp. 690-691, September 1979

[8] Microsoft Windows NT homepage, URL: `http://www.microsoft.com/ntserver/`

[9] Paas, S. M.; Scholtyssik, K.: *Efficient Distributed Synchronization within an all-software DSM system for clustered PCs.* 1st Workshop Cluster-Computing, TU Chemnitz-Zwickau, November 6-7, 1997

[10] Speight, E., Bennett, J. K.: *Brazos: A Third Generation DSM System*, Proc. of the USENIX Windows NT Workshop, Seattle, 1997

[11] Thitikamol, K.; Keleher, P.: *Multi-Threading and Remote Latency in Software DSMs*. In: 17th International Conference on Distributed Computing Systems, May 1997

[12] Sunsoft Solaris homepage, URL: `http://www.sun.com/software/solaris/`

[13] SVMlib Homepage, URL: `http://www.lfbs.rwth-aachen.de/~sven/SVMlib/`

[14] UNIX to NT resource center, URL: `http://www.nentug.org/unix-to-nt/`

[15] Woo, S. C.; Moriyoshi Ohara, M.; Torrie, E.; Singh, J. P., and Gupta, A.: *The SPLASH-2 Programs: Characterization and Methodological Considerations.* In Proc. of the 22nd International Symposium on Computer Architecture, pp. 24-36, Santa Margherita Ligure, Italy, June 1995