The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

Win32 API Emulation on UNIX for Software DSM

Sven M. Paas, Thomas Bemmerl, Karsten Scholtyssik
RWTH Aachen, Lehrstuhl für Betriebssysteme

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Win32 API Emulation on UNIX for Software DSM

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

Abstract*. This paper presents a new Win32 API 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 systems programming like exception handling, virtual memory management, Windows NT multithreading / synchronization and the WinSock API for networking. As a case study, we ported the all-software distributed shared memory (DSM) system SVMlib - consisting of about 15.000 lines of C++ code written natively for the Win32 API - to Sun Solaris 2.5 with absolutely no source code changes.*

1 Introduction

While there exist numerous products and toolkits designed for porting software from UNIX[1] to Windows NT on source code level (like OpenNT from Softway Systems, Inc. [14], or other tools [17]), much less effort has been conducted to provide a toolkit to port Windows NT applications, especially those using the Win32 API directly, to UNIX. Apart from some very expensive commercial products in this area like *Wind/U* from Bristol Technology, Inc. [2] or *MainWin XDE* from MainSoft Corp. [9], a small low cost solution is not known to the authors. With the emerging importance of Windows NT [10], 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 [15], 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:

- *Windows NT Multithreading and Synchronization*. 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.
- *Windows NT Exception Handling*. Win32 supports user level handlers to catch special exceptions as well as certain error handling routines. These functions form another group to be supported.
- *Networking*. This group concerns networking, which includes the complete WinSock API (restricted to the TCP/IP protocol family, however).

Naturally, an emulation layer firstly has to support the basic data types found in the various C++ header files of 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 Multithreading and Synchronization

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

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

nt2unix uses the *Standard Template Library* (STL) type `map` to store the above information for each thread. The entries in the map are indexed by the Windows NT handle of the thread. Accesses to this map are protected by a special lock object of class `CriticalSection`, which is a comfortable wrapper around the Windows NT `CRITICAL_SECTION` type:

```
class CriticalSection {
  public:
    CriticalSection::CriticalSection() {
      InitializeCriticalSection(&cs);
    }
```

```
    CriticalSection::~CriticalSection() {
      DeleteCriticalSection(&cs);
    }
    inline void CriticalSection::enter() {
      EnterCriticalSection(&cs);
    }
    inline void CriticalSection::leave() {
      LeaveCriticalSection(&cs);
    }
  protected:
    CRITICAL_SECTION cs;
};

struct ThreadInfo {
  ThreadInfo::ThreadInfo() {
    ThreadInfo::init(THREAD_RUNNING);
  }
  ThreadInfo::ThreadInfo(DWORD aState) {
    ThreadInfo::init(aState);
  }
  inline void ThreadInfo::init(DWORD aState) {
    state = aState;
    suspendCount = 0;
    exitCode = 0;
    threadHasBeenResumed = FALSE; // see below
  }
  DWORD suspendCount;
  DWORD state;
  DWORD exitCode;
  volatile BOOL  threadHasBeenResumed;
  // A special flag to synchronize
  // SuspendThread() / ResumeThread()
};

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

Important 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**() and **thr_resume**(). However, this is not the case, since there is a lost signal problem to be avoided when a thread suspends. The situation using the POSIX thread API is even worse, because there are no functions available for resuming or suspending threads anyway.

To understand this, we have a deeper look at our implementation of **SuspendThread**() using the Solaris thread API. 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 counter 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**() looks 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;
}
```

Other caveats occur in order to support synchronization function like **`EnterCriticalSection`**() and **`LeaveCriticalSection`**(), because under Windows NT the `CRITICAL_SECTION` objects can be acquired recursively (that is, a lock owning thread may acquire the same lock without deadlocking), while Solaris / POSIX thread mutexes are not. The solution to this problem is again to reinvent the wheel and try to emulate this behavior. We use the standard Windows NT type

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;
    HANDLE LockSemaphore;
    DWORD Reserved;
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

for each critical section object and its thus possible to keep track of recursive lock acquires and releases:

```
WINBASEAPI VOID WINAPI EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection) {
  thread_t me = thr_self();
  if (lpCriticalSection->OwningThread ==
    (HANDLE)me) {
    // I have the lock.
    // This cannot be a race condition.
    lpCriticalSection->RecursionCount++;
    return;
  }
  if(mutex_lock((mutex_t *)(lpCriticalSection
    ->LockSemaphore))) {
    DBG("mutex_lock() failed"); return;
  }
  // got it. I must be the first thread:
  if (lpCriticalSection->RecursionCount) {
    DBG("RecursionCount != 0"); return;
  }
  lpCriticalSection->RecursionCount = 1;
  lpCriticalSection->OwningThread = (HANDLE)me;
  return;
}
```

If the thread acquiring a lock is the same thread already owning the lock, the recursion counter is incremented. This cannot be a race condition, because no other thread can change this value at the time the lock is blocked. Otherwise, **`mutex_lock`**() (or **`pthread_mutex_lock`**() in the POSIX version) is called. If the lock is successfully acquired, the recursion counter must be 0 and is set to 1 afterwards.

```
WINBASEAPI VOID WINAPI LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection) {
    thread_t me = thr_self();
    if (lpCriticalSection->OwningThread ==
      (HANDLE)me) {
      lpCriticalSection->RecursionCount--;
      if(lpCriticalSection->RecursionCount < 1) {
        lpCriticalSection->OwningThread =
          (HANDLE)0xFFFFFFFF;
        if(mutex_unlock((mutex_t *)
          lpCriticalSection->LockSemaphore))
          DBG("mutex_unlock() failed");
      }
    } else
      DBG("not lock owner");
  return;
}
```

When leaving a critical section, the recursion counter is decremented if the call was recursive. If the counter is 0 no thread owns the lock anymore, hence **`mutex_unlock`**() (or **`pthread_mutex_unlock`**() in the POSIX version) must be called. It is an error if the caller of **`LeaveCriticalSection`**() was not the owner of the lock.

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;
```

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

The virtual base address for the mapping is stored in `lpBaseAddress`, while the size of the mapping object in bytes is stored in `dwNumberOfBytes`. The handle of the mapped file and / or its file name are stored in `hFileMappingObject` and `FileName[]`, respectively. The above `struct` is allocated for a specific mapping by calling our emulation of **`CreateFileMapping`**() or **`CreateFileMappingA`**(), respectively. It is deallocated if the `refcnt` counter keeping track of the open references to the mapping for the mapping is 0. Using a STL-style `vector` of mappings, Windows NT mapping is 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("Ignoring dwFileOffsetHigh");
  // Filter the protection bits
  // and mapping flags ...
  prot = dwDesiredAccess & FILE_MAP_ALL_ACCESS;
  flags = ((dwDesiredAccess & FILE_MAP_COPY) ==
    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;
}
```

The similar function **`MapViewOfFile`**() is now easily implemented by calling **`MapViewOfFileEx`**() with the last parameter *lpBaseAddress* set to 0. The mentioned `refcnt` for each mapping is decremented by a call to **`UnMapViewOfFile`**(). However, the above implementation has a 4 GB limit concerning the maximum size of the mapped file, because this is the maximum file mapping size possible under Solaris.

The memory access protection bits of a file mapping under Windows NT have more a less equivalent values under UNIX. However, not all bit masks are supported, namely `PAGE_GUARD` and `PAGE_NOCACHE`:

Windows NT Protection Bits	UNIX Bits
PAGE_READONLY	PROT_READ
PAGE_READWRITE	(PROT_READ | PROT_WRITE)
PAGE_NOACCESS	PROT_NONE
PAGE_EXECUTE	PROT_EXEC
PAGE_EXECUTE_READ	(PROT_EXEC |PROT_READ)
PAGE_EXECUTE_READWRITE	(PROT_EXEC | PROT_READ | PROT_WRITE)
PAGE_GUARD	n/a
PAGE_NOCACHE	n/a

2.3 Windows 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`**().

Because the first method is a proprietary language extension, only 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:

Windows NT `EXCEPTION_*` Code	UNIX Signal
ACCESS_VIOLATION	SIGSEGV
FLT_INVALID_OPERATION	SIGFPE

Windows NT EXCEPTION_* Code	UNIX Signal
ILLEGAL_INSTRUCTION	SIGILL
IN_PAGE_ERROR	SIGBUS
SINGLE_STEP	SIGTRAP

The emulation of Windows NTs exception handling requires carefully converting UNIX-style types like `siginfo_t` and `ucontext_t` to a Windows NT-style `struct EXCEPTION_POINTERS`. To fill in this `struct`, the stack frame of the signal handler method must be examined, which is very system dependent.
For example, the page fault info is extracted in a SIGSEGV handler for Solaris SPARC (`__SPARC`), Solaris x86 (`__X86`) and Linux x86 (`__LINUXX86`) in the following way:

```
switch (sig) {
case SIGSEGV:
    // A segmentation violation.
    ExceptionInfo.ExceptionRecord->
      ExceptionCode = EXCEPTION_ACCESS_VIOLATION;
    ExceptionInfo.ExceptionRecord->
      ExceptionInformation[0] =
#if defined(__SPARC)
       (*(unsigned *)((ucontext_t*)uap)
         ->uc_mcontext.gregs[REG_PC] & (1<<21));
#elif defined(__X86)
       (((ucontext_t*)uap)->
         uc_mcontext.gregs[ERR] & 2);
#elif defined(__LINUXX86)
       stack[14] & 2;
#endif
    if (ExceptionInfo.ExceptionRecord->
      ExceptionInformation[0])
      ExceptionInfo.ExceptionRecord->
        ExceptionInformation[0] = 1;
        // 1 == write access
    ExceptionInfo.ExceptionRecord->
      ExceptionInformation[1] =
#ifdef __LINUXX86
       stack[22];
#else
       (DWORD)sip->si_addr;
#endif
    break;

    // other signals processed here ...
}
```

If a `SIGSEGV` is caught, the exception type is set to `EXCEPTION_ACCESS_VIOLATION`. In the next statements, the type of the fault (read or write) as well as the faulting address must be extracted from the stack. Under Solaris SPARC, the type of the fault is coded in bit 21 of the `REG_PC` register, while under Solaris x86, bit 2 of the `ERR` register contains this information. Under Linux x86, bit 2 of the `stack` at position 14 stores this bit of `ERR` according to the Linux 2.0 kernel source.
The faulting address under Solaris is located under `sip->si_addr`, where `sip` of type `siginfo_t*` is the second parameter of the signal handler function installed. Under Linux, the value is found at position 22 of the signal handler stack.
Of course, this code is not portable and must be implemented again for each UNIX derivative.

2.4 Networking

The standard protocol family available under UNIX is TCP/IP. With nt2unix, we map the WinSock API with respect to this protocol to the standard BSD sockets API. The main difference between the Windows NT and the BSD socket API is due to some new Windows NT data types and definitions:

```
typedef int                SOCKET;
#define INVALID_SOCKET     (SOCKET)(-1)
#define SOCKET_ERROR       (-1)
```

Additionally, Windows NT defines the Windows Sockets (WinSock) API error codes (`WSA*`). The only real problem while emulating the WinSock API under BSD was found for the **`select`**() call. This function has the prototype

```
int select(int nfds, fd_set FAR *readfds,
           fd_set FAR *writefds,
           fd_set FAR *exceptfds,
           const struct timeval FAR * timeout);
```

A source of hard to find programming errors is that the `fd_set` data type is usually implemented as a bit mask under UNIX, while Windows NT implements this data type as an ordinary array. That is the reason why Windows NT ignores the first parameter `nfds` which defines the highest bit to be scanned while waiting for pending input. That is, you can unfortunately write Windows NT code using **`select`**() which doesnot run under BSD.

2.5 Summary

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

	Win32 Functions emulated	Emulation is based on
Multi-threading	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()

	Win32 Functions emulated	Emulation is based on
Thread Synchronization	InitializeCritical-Section() DeleteCriticalSection() EnterCriticalSection() LeaveCriticalSection()	mutex_init() mutex_destroy() mutex_lock() mutex_unlock()
Thread Local Storage (TLS)	TlsAlloc() TlsGetValue() TlsSetValue() TlsFree()	thr_keycreate() thr_getspecific() thr_setspecific() pthread_key_ delete()
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	SetUnhandled-ExceptionFilter() GetException-Information() UnhandledExcep-tionFilter()	sigaction()
Miscellaneous	GetSystemInfo() GetComputerName() QueryPerformance-Frequency() QueryPerformance-Counter()	sysinfo() gethostname() - gettimeofday()

3 A Case Study: SVMlib

3.1 Overview

SVMlib [11, 16] (*Shared Virtual Memory Library*) is an all-software, page based, user level shared virtual memory [1] subsystem for clusters of Windows NT workstations. It is one of the first (among [7] and [12]) 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 users 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 the Dolphin implementation [3] of the *Scalable Coherent Interface* (SCI) [5].

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. Different synchronization primitives such as 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* [8] model. This model is supported by single writer as well as multiple writer protocols. Secondly, the distributed lock based *scope consistency* [6] is implemented.

3.2 Design Issues

When designing an 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 the primary platform. Figure 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 the user to create and destroy primitives for distributed process synchronization - mutexes as well

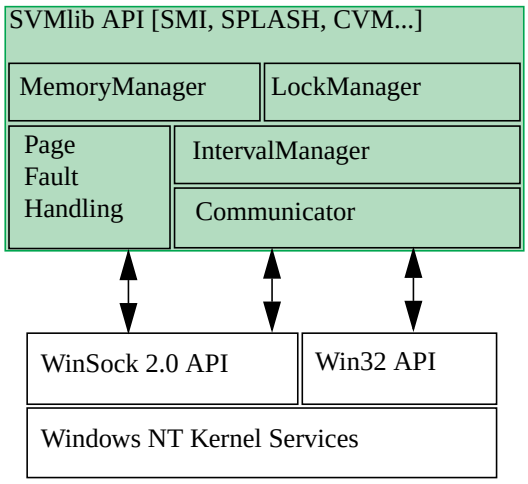

Figure 1: SVMlib layers

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.

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) [4], the macro interface of *Stanford Parallel Applications for Shared Memory* (SPLASH) [18] and the *Coherent Virtual Machine* (CVM) [13]. 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. Though the development of nt2unix was in fact driven by our goal to port SVMlib to UNIX, 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 characterizing the performance of the SVM 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 Windows NT Server and NT Workstation (WS) 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 using nt2unix, respectively. As mentioned, under UNIX, the memory exception handling mechanism of Windows NT is emulated by catching the `SIGSEGV` signal.

	Super-SPARC, 50 MHz	**Pentium, 133 MHz**	**Pentium Pro, 200 MHz**
Windows NT 4.0 Server / WS	-	28 µs	19 µs
Solaris 2.5.1 & nt2unix	135 µs	92 µs	48 µs

- *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 [13] 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

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)
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

It is clear that the above measurements are not sufficient to determine the overall performance of nt2unix, but they show reasonable results with respect of key functions used to implement SVMlib on Windows NT: multithreading, networking and exception handling.

3.4 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. The first version of nt2unix was developed and tested on Solaris 2.5 for SPARC and Intel Processors, respectively. At the moment, we are extending the implementation to support more generic UNIX platforms and POSIX interfaces:

- The current version supports the POSIX thread API additionally to the Solaris thread API. This required slightly different implementation of **`ResumeThread`**() and **`SuspendThread`**(), because POSIX doesnot include functions equivalent to Solaris **`thr_resume`**() and **`thr_suspend`**().
- We have a running Linux/x86 version of nt2unix using a POSIX thread library. We found that especially the exception handling is very system dependent, because the signal handler stack frame has to be inspected to extract the detailed exception information.

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 acceptable. The complete source code of the nt2unix library is available upon 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] Bristol Technology Inc., URL: `http://www.bristol.com/`

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

[4] 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.

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

[6] 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

[7] 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

[8] 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

[9] MainSoft Corp., URL: `http://www.mainsoft.com/`

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

[11] 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

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

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

[14] Softway Systems, Inc. URL: `http://www.softway.com/`

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

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

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

[18] 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

[1]UNIX is a registered trademark of The Open Group licensed exclusively in conjunction with a brand program.