Win32 API Emulation on UNIX for Software DSM

Sven M. Paas, Thomas Bemmerl, Karsten Scholtyssik
Lehrstuhl für Betriebssysteme, RWTH Aachen, Germany
http://www.lfbs.rwth-aachen.de/
contact@lfbs.rwth-aachen.de

Agenda:

• Background
• Our approach: emulating a reasonable Win32 API subset
• Implementation details of nt2unix (multithreading, memory mapped I/O, ...)
• A case study: SVMlib: Shared Virtual Memory Library
• Conclusions
Background

The Problem:

• Given a **console** application written in C / C++ for Win32;
  --> Visual C++ 5.0 + STL, Windows NT 4.0 / Windows 95
• Compile (and execute) the same code on a UNIX system
  --> gcc 2.8.1 + STL, Solaris 2.6 [SPARC/x86], Linux 2.0 [x86]

Available Solutions:

• *Wind/U* (Bristol Technology, Inc., http://www.bristol.com/)
• *MainWin XDE* (MainSoft Corp., http://www.mainsoft.com/)
• *Willows Twin API* (Canopy Group, http://www.willows.com/)
A reasonable Win32 Subset

• NT Multithreading
  Creating / Destroying / Suspending / Resuming preemptive threads
  Synchronization and Thread Local Storage (TLS) functions;

• Virtual Memory (VM) Management
  Allocating / Committing / Protecting VM on page level
  Memory Mapping I/O, File Mapping

• NT Structured Exception Handling (SEH)
  User Level Page Fault Handling by SEH

• Networking using WinSock
  Windows Sockets API for TCP/IP
Windows NT Multithreading

Creating a Thread under NT:

WINBASEAPI HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter, DWORD dwCreationFlags,
    LPDWORD lpThreadId);

with

typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(
    LPVOID lpThreadParameter);

typedef PTHREAD_START_ROUTINE LPTHREAD_START_ROUTINE;
UNIX Multithreading

Creating a Thread using POSIX API:

```c
int pthread_create(
    pthread_t *new_thread_ID,
    const pthread_attr_t *attr,
    void *(*start_func)(void *), void *arg);
```

... and using the Solaris Thread API:

```c
int thr_create(void *stack_base, size_t stack_size,
    void *(*start_func)(void *), void *arg, long flags,
    thread_t *new_thread_ID);
```

--> we must ignore LPSECURITY_ATTRIBUTES.
(like Windows 95 / 98 does)
NT Thread Synchronization

Problems:

- Suspending / Resuming Threads is **not** possible within the POSIX Thread API! (-> SuspendThread(), ResumeThread())
- This fact implies that some Win32 thread concepts are hard to implement **efficiently** within POSIX environments:

```c
struct ThreadInfo {
    DWORD state, suspendCount, exitCode;
#ifdef __POSIX_THREADS__
    pthread_cond_t cond, pthread_mutex_t mutex;
#else
    volatile BOOL threadHasBeenResumed;
#endif
};
```
Virtual Memory (VM) Management

Emulating a Windows NT File Mapping Object:

```c
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;
```
NT Structured Exception Handling

Two methods:

- by embracing code with a `try{} ... except() {}` block;
- by installing a user level exception handler by calling `SetUnhandledExceptionFilter()`. 

Translation of NT Exception Codes to UNIX signals:

<table>
<thead>
<tr>
<th>Windows NT EXCEPTION_* Code</th>
<th>UNIX Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_VIOLATION</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>FLT_INVALID_OPERATION</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>ILLEGAL_INSTRUCTION</td>
<td>SIGILL</td>
</tr>
<tr>
<td>IN_PAGE_ERROR</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>SINGLE_STEP</td>
<td>SIGTRAP</td>
</tr>
</tbody>
</table>
Catching Page Faults

1st problem: where was the fault?

```c
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
```
Catching Page Faults (cont‘d)

2nd problem: what was the reason for the fault?

```c
if (ExceptionInfo.ExceptionRecord->
    ExceptionInformation[0])
    ExceptionInfo.ExceptionRecord->
    ExceptionInformation[0] = 1;
    // 1 == write access; 0 == read access
    ExceptionInfo.ExceptionRecord->
    ExceptionInformation[1] =
#ifdef __LINUXX86
    stack[22];
#else
    (DWORD)sip->si_addr;
#endif
    break;
    break;

    // other signals processed here ...
}
```
TCP/IP Networking using WinSock

Ideas:
- Restrict WinSock 2.0 to BSD Socket API
- Translate data types, definitions, and error codes

For example:

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

Pitfalls:
- some types are hard to map (e.g. `fd_set`)
- WinSock‘s `select()` is not BSD `select()`!
A Case Study: SVMlib

SVMlib: Shared Virtual Memory Library

- all software, user-level, page based
- about 15,000 lines of (Visual) C++ code, natively for Win32
**SVMLib Performance (1)**

**Page Fault Detection Time:**

<table>
<thead>
<tr>
<th></th>
<th>Super-SPARC, 50 MHz</th>
<th>Pentium, 133 MHz</th>
<th>Pentium Pro, 200 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT 4.0 Server / WS</td>
<td>-</td>
<td>28 µs</td>
<td>19 µs</td>
</tr>
<tr>
<td>Solaris 2.5.1 (native)</td>
<td>105 µs</td>
<td>70 µs</td>
<td>40 µs</td>
</tr>
<tr>
<td>Solaris 2.5.1 &amp; nt2unix</td>
<td>135 µs</td>
<td>92 µs</td>
<td>48 µs</td>
</tr>
</tbody>
</table>

--> UNIX Signal handling is **expensive**.
SVMLib Performance (2)

Page Fault Handling Times:

<table>
<thead>
<tr>
<th>Nodes</th>
<th>R / W / Avrg Fault Time [ms]</th>
<th>R / W / Avrg Fault Time [ms]</th>
<th>R / W / Avrg Fault Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>CVM on Solaris (Sun SS20)</strong></td>
<td><strong>SVMLib on nt2unix (Sun SS20)</strong></td>
<td><strong>SVMLib on Windows NT (Intel Pentium 133)</strong></td>
</tr>
<tr>
<td>2</td>
<td>11.3 / 0.8 / 4.4</td>
<td>4.5 / 1.3 / 2.2</td>
<td>3.4 / 1.1 / 1.8</td>
</tr>
<tr>
<td>3</td>
<td>12.0 / 0.8 / 5.8</td>
<td>4.6 / 1.8 / 2.7</td>
<td>3.4 / 1.4 / 2.3</td>
</tr>
<tr>
<td>4</td>
<td>16.7 / 0.9 / 7.1</td>
<td>4.9 / 1.8 / 3.1</td>
<td>4.0 / 1.5 / 2.4</td>
</tr>
</tbody>
</table>

Test Application: FFT
Conclusions

• Win32 API Emulation under UNIX is possible.

• If the Emulation is „application driven“, it can be implemented within finite time (3 MM for SVMlib);

• nt2unix is a reasonable first step to develop portable low level applications.

Next Steps:

• More complete implementation of Win32 base services;

• More applications (NT Services <-> UNIX Daemons)
Further Information

nt2unix Project Homepage:

http://www.lfbs.rwth-aachen.de/~sven/nt2unix/

SVMlib Project Homepage:

http://www.lfbs.rwth-aachen.de/~sven/SVMlib/

E-Mail:

contact@lfbs.rwth-aachen.de

Phone/Fax:

+49-241-7634, +49-241-8888-339