Porting Windows NT ® Applications to Linux

By Jeremy Allison

Development Team

email: jra@samba.org

Windows Porting - What is possible (and what isn't).

- Windows GUI client applications are a nightmare to port.
  - The Win32 GUI API is horribly complex, with many subtle interdependencies.
  - To port a Windows GUI application, without too much change, winelib from the WINE project is the best hope.
    - This is the route Corel have taken.
    - Best possible solution if the application is large and must keep running on Windows.
  - Qt, GTK are cross platform GUI toolkits. wxWindows based on GTK also cross platform.
    - Usable if the GUI application is small and can be re-written.
  - Windows GUI programs are in a world of pain on Linux :-).

Windows Porting - what is reasonable.

- Windows NT service control code is reasonably easy to port.
  - Services have defined start/stop interfaces - convert these to SYSV init.d scripts (in RedHat).
  - "Suspend" interface maps to SIGTSTOP (can be caught).
  - "Configure" interface maps to SIGHUP (used by convention to re-read configure files for a daemon).
- Each of these control notifications is run under a different thread on NT.
  - Default model on Linux is to use a signal handler, set a variable that is checked in main code.

Multi-Threaded vs. Multi Process

- Windows NT code tends to be multi-threaded rather than multi-process (traditional UNIX model).
  - Win32 threads have significant differences from UNIX pthreads model. Study a good pthreads book.
  - Win32 synchronization primitives (Mutexes, Events, Semaphores) all can be built using pthreads primitives.
  - Beware of different semantics between similar looking primitives. For example, Win32 Mutexes are recursive, pthreads mutexes are not.
  - Beware of subtle dependencies on Win32 semantics.

Multi-Threaded vs. Multi-Process (continued).

- For performance reasons, Win32 multi-threaded code may better be re-architected as multi-process code on Linux.
  - Linux much better at handling multi-processes. Process creation is much cheaper under Linux than NT (about the same as thread creation on NT).
- Multi-process code easier to debug under Linux than multi-threaded code.
- Multi-process code more robust against programmer error than multi-threaded code (example, Samba vs. NT SMB server).

Win32 to UNIX : the easy functions.

- Win32 sockets map almost directly to UNIX BSD sockets.
  - Only difference is fd_set code. Win32 uses an array of HANDLES rather than a bitmask.
- MapViewOfFile maps easily into mmap and friends.
- Most file I/O functions map easily (ReadFile -> read, WriteFile -> write). Better performance may come from mapping to stdio functions.
- File locking maps to fcntl (POSIX) locking.
  - POSIX locking is more flexible.

Win32 to UNIX : the easy functions (continued).

- Win32 LoadLibrary and associated functions can be mapped directly into dlopen() and friends.
  - Differences in semantics (private linking of C library data) not usually an issue in most code.
- Win32 eventlog functions are easily mapped into UNIX syslog functions.
- CreateProcess (monster function) maps easily into fork()/ exec() pair.
  - UNIX functions more flexible. Makes duplicating needed CreateProcess semantics easy.

Win32 to UNIX : more complex mappings.

- Win32 uses WaitforMultipleObjects to synchronize all events.
  - Limited to 64 HANDLES, but can be HANDLES to any waitable object.
- Under UNIX, poll() may be best mapping.
  - However poll() only waits on file descriptors.
  - Common trick is to use signal handlers that write messages down a pipe file descriptor to map other waitable objects to a poll().
  - Beware of differences between slow (interruptible via signals) and fast system calls. Win32 has no similar concept.

Win32 to UNIX : more complex mappings (continued).

- Win32 message pump code may be replaced with SystemV message queues (or POSIX message queues once these become standard).
  - MsgWaitforMultipleObjects() is probably best replaced with poll(), using UNIX domain sockets to transfer the messages, rather than SYSV message queues.
  - SYSV functions suffer from poor integration with other synchronization mechanisms (as they are not file descriptor based).
- Win32 heap code may be discarded - replace with simple malloc.
  - Different malloc libraries available under UNIX for different uses.

Win32 to UNIX : security considerations.

- Win32 has an amazingly complex and over engineered security model.
  - Every object can be separately secured.
  - Almost no Win32 code uses it.
- UNIX relies on file system security to secure shared objects.
  - Much simpler to use.
  - Several subtle pitfalls. O_EXCL to stop race conditions as an example.
  - "root" daemons are system level code.
- In practice most Win32 code will ignore security (e.g.. Microsoft applications).

Windows Porting - here be dragons !

- Any code using MFC will be difficult to port.
  - MFC classes are designed to tie together "model" and "view". Roach motel class library.
  - If service code is just using MFC collection classes, then porting to C++ STL standard is recommended.
  - Any code tied to Windows message pump processing will have to be re-written.
- COM/DCOM code must be replaced.
  - COM on Linux libraries exist - third party, binary only.
  - Options include ONC/RPC or Corba libraries (all available with source code).

Windows porting - more dragons.

- I/O completion ports and asynchronous I/O under Win32 is the most difficult area to address.
  - Win32 code using this will be considered to be performance sensitive.
  - No identical concepts under Linux.
- UNIX async I/O comes closest.
  - Linux implementation currently done in user space - not high performance.
  - SGI has donated Linux kernel async I/O. May go into kernel 2.4 ?
- Older SIGIO async I/O may work.
  - Measure performance to be sure (means more complex code under UNIX than Win32)

Windows porting - things to throw away.

- Windows structured exception handling.
  - g++ on Linux will do C++ exception handling.
  - No Linux equivalent to Windows C based __try keyword.
  - Most code that depends on this is attempting to catch top level errors to display a dialog box. Linux does useful core dumps instead.
- Microsoft "additions" to the C language.
  - Things like __declspec(), __dllexport(), __dllimport() and other monstrosities.
- Thread local storage must be done explicitly, rather then using the Microsoft keyword.

Weeding out Windows-isms.

- Win32 has very poor abstract type definitions.
  - Most functions use DWORD or other types specified as an absolute size.
- Under UNIX, replace these with correct abstract types (eg. off_t for file sizes, size_t for string lengths).
- Win32 is tied to 32 bit semantics. Porting to a 64 bit UNIX platform takes great care (but doing so will fix many bad assumptions).
- Win32 little-endian by definition.
  - All data file reading/writing code must be fixed to work on big endian processors.

Porting Summary.

- Bear in mind other UNIXs than Linux when porting.
  - Making code portable across POSIX platforms is worthwhile in the long run (not tied to single platform).
- Weed out non-portable APIs.
  - Study POSIX specifications when deciding what to APIs to use.
- Port with 64 bit platforms in mind.
  - 32 bit platforms (such as Win32) will become legacy very quickly.

References.

- Win32 API definitions.
  - Check out MSDN site at microsoft.com.
  - "Advanced Windows Programming" : Jeffrey Richter.
- Cygnus/RedHat : Cygwin32 product.
  - An excellent example of how to map POSIX -> Win32.
  - Allows your new UNIX port to continue running on Windows !
- POSIX Spec: IEEE document.
- UNIX pthreads books :
  - "Programming with POSIX Threads": Dave Butenhof
  - "Pthreads Programming": Dick Buttlar & Jacqueline Farrell (O'Reilly book).