Porting Windows NT ® Applications to Linux

By Jeremy Allison

Development Team

email: jra@samba.org

Windows Porting - why do it ?

- Portability. Win32 API is not usable on anything other than Windows (valiant attempts like WINE notwithstanding).
- POSIX API's are stable and common across many different versions of UNIX.
  - Even more so than Win32. Originally touted as a "common" Windows API there are now over 5 different versions.
- Win32 API is an incoherent design. Error returns differ across functions, few abstract data types (hard to move to 64 bit). Undergoes rapid changes.

Windows Porting - why do it ?

- Financial gain : Linux is the fastest growing server OS. Large categories of software are as yet uncolonised on Linux.
- No competition from the platform vendor : "The Windows API supports ISV's in the same manor as a rope supports a man about to be hung" (Andrew Schulman).
- Customizable : Open Source allows the exact behavior of the OS to be determined. Puts application vendors in charge of their own destiny....

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 rewritten.
  - 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.
- Two possibilities for network services.
  - Stand-alone daemon - always running.
  - Program managed by inetd superserver - much easier to use than NT style code.
- For non-network services then stand-alone daemon is the only choice.

Stand-alone daemons

- The only choice for a non-network server.
- Most similar to an NT service.
- Usually run as "root" (equivalent to NT LocalSystem account).
  - root has more abilities than LocalSystem, can impersonate users at will.
  - If you can, design so that you lose root privileges as soon as possible. Greatly adds to security.
- No registry on UNIX. Store configuration information in files - ASCII editable files are always preferred.

Stand-alone daemon code

- Good examples are Samba (smbd/nmbd) and Apache.
- Typically one process per connected client.
- Managed by start/stop scripts held in /etc/rc.d/init.d.
  - Dependencies managed by run levels and numeric ordering of script names.
  - Look into /etc/rc.d/rc3.d and /etc/rc.d/rc5.d for good examples.
  - S<number>name is start script.
  - K<number>name is kill (shutdown) script.

Controlling stand-alone daemons

- No API equivalent to "ControlService()". Use asynchronous signals instead.
- ControlService() notifications are run under a different thread on NT.
  - Default model on Linux is to use a signal handler, send a notification that is checked in main code.
    - SERVICE_CONTROL_STOP -> send SIGTERM (not SIGKILL).
    - SERVICE_CONTROL_PAUSE -> send SIGTSTP (not SIGSTOP).
    - SERVICE_CONTROL_CONTINUE -> send SIGCONT.
    - No method of performing SERVICE_CONTROL_INTERROGATE -> possibly map to SIGHUP.
    - Causes most daemons to reread configure files and reconfgure themselves.

Controlling stand-alone daemons

- UNIX has richer semantics for grouping processes. Signals can be used to control a related group of processes called a "session".
- Examine code sample 1.
- UNIX threading is done via pthreads.
- Examine code sample 2.
  - Performs stand-alone daemon control in a threaded manor (easier to understand for NT programmers).
  - WARNING ! This code has a dependency on Linux pthreads (signals need to be sent to the correct thead id, not parent process id).

Inetd daemons

- If possible, use the inetd super-server to start your daemon. Extra security (tcpd) is available.
- inetd can accept TCP or UDP connections on behalf of your daemon and will invoke you with stdin, stdout, stderr set to the network socket.
  - Beware of stderr printf calls from library functions...
- Inetd can invoke a new daemon per incoming connection, or invoke one daemon and wait until it finishes.
- Configuration file in /etc/inetd.conf. Example line:
  - swat stream tcp nowait root /usr/local/samba/bin/swat swat

Inetd daemon code

- To determine if your daemon has been started by inetd, attempt a socket operation (getsockopt()) on file descriptor zero. If it succeeds, you were started from inetd.
- Install issues are more difficult for inetd daemons.
  - /etc/inetd.conf file listens on ports defined in /etc/services.
  - service lookups (the getservbyname() call) may be served via a network lookup (NIS or NISPLUS).
- Inetd daemon code less suited to high performance.

Asynchronous signals

- These are really "software interrupts". Can be used to break into "slow" system calls in the case where threads would be used in NT.
- See sample3.c for an example of using the SIGALRM signal for this.
- SetConsoleCtrlHandler() is the closest NT concept, however this spawns a new thread internally to deal with the notifications.
- Signals and POSIX pthreads don't mix too well. A separate thread handler is best used (sample2.c) in the UNIX threaded case.

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).

Creating processes

- Win32 CreateProcess() is a monster function. Far too many parameters (10) and 8 configurable flags that modify behavior.
- UNIX has fork() - returns twice (subtle point). Followed by exec() it allows the same semantics as CreateProcess().
  - Having this functionality in two separate functions allows application specific modifications in the process environment before invoking the child process.
- In order to create a process under a different user context, Win32 requires a plaintext password for that user (and a system privilege).

Waiting for processes to die

- Win32 uses "standard" wait function WaitForXXObjects() to synchronize parent with child death.
  - Somewhat limited - only 64 HANDLES can be used with WaitForMultipleObjects().
- POSIX uses asynchronous signals to signal child death.
  - Parent can be set to ignore child death (no signal) - equivalent to closing the child process handle on NT.
  - Child status can be returned using the waitpid() call.

Creating threads

- Win32 threading model is well defined. POSIX threads were an additional interface and are less well integrated into the system.
- CreateThread() or _beginthreadex() may be replaced by pthread_create().
- Older POSIX interfaces are not MT safe (eg.gethostbyname()). Some interfaces have been extended by adding a _r suffix (to signify reentrant).
  - This is similar to the Win32 Ex suffix.
- Don't assume calls are MT safe unless explicitly documented as such.

Win32 to UNIX : sockets

- Win32 sockets map almost directly to UNIX sockets.
- Significant difference is fd_set code. Win32 uses an array of HANDLES rather than a bitmask.
- Win32 socket HANDLES need to be created as non-overlapped in order to use them with default ReadFile()/WriteFile().
  - UNIX sockets are just ordinary file descriptors and can be treated exactly as such. read()/write() work as expected. inetd depends on this.
- The "blocking hook" concept doesn't exist under POSIX - was an old 16-bit Windows method.

Win32 to UNIX : memory mapped files

- UNIX mmap is easier to use than Win32 memory mapping.
- One system call to map a file region (mmap()), one call to remove the mapping (munmap()).
  - Auxiliary calls to flush views etc.
- Win32 calls, although adding an extra API layer (a "File Mapping Object") are functionally identical.
  - So much so POSIX file mapping is implemented in terms of Win32 file mapping in Cygwin32, a POSIX emulation layer on top of Win32.

Win32 to UNIX : file locking

- File locking under Win32 is much easier to use than POSIX.
- Semantics are also different, Win32 file locking is mandatory, POSIX locking is advisory.
- See sample4.c for how to use fcntl locking.
- POSIX has much richer locking semantics. Lock ranges can be merged and split, and lock ranges can be upgraded (read -> write lock) and downgraded (write -> read lock).
- Win32 locking semantics are not precisely known.

Win32 to UNIX : event logging

- UNIX eventlogging API's are easier to use than the Win32 ones.
  - openlog()/syslog()/closelog() are the only relevant functions.
- syslog() is a varargs function allowing printf style formatting of log messages.
- The clean architecture split (the logging functions just write down a local pipe to a separate logging daemon) allows easy remoting of logs, without the application needing to be aware.
- Logging is controlled by the syslog daemon, and the /etc/syslog.conf file.

Win32 to UNIX : input/output

- Win32 has several different I/O types.
  - Synchronous, Asynchronous (Overlapped) and I/O completion port.
  - Using I/O completion ports are complex but can give high thoughput.
- UNIX has synchronous and asynchronous (old style SIGIO and newer style POSIX aio_XX functions).
  - Currently the aio_read()/aio_write() functions in Linux 2.2 are emulated using threads created by glibc.
  - Win32 Overlapped code maps reasonably well to the newer POSIX.4 aio_XX calls.

Win32 to UNIX : I/O completion ports

- Currently no UNIX has a mechanism similar to Win32 I/O completion ports.
  - I/O completion ports allow a limited number of worker thread to keep an asynchronous I/O handle busy.
  - Whenever an I/O request completes one of the worker threads is woken to process the request.
- The Linux 2.4 kernel will have a mechanism to reproduce I/O completion ports (queued POSIX real-time signals with sigwaitinfo()).
  - The signal number chosen will represent the completion port.

Win32 to UNIX : shared libraries

- Win32 shared library functions : LoadLibrary(), GetProcAddress(), FreeLibrary() map easily onto the UNIX dlopen(), dlsym() dlclose() functions.
- UNIX elf shared object format is more powerful than the Win32 COFF based format.
  - UNIX shared libraries can link back to symbols in the loading code, Win32 shared libraries must be complete when created.
- Concepts map reasonably well, however when running root daemons be careful to use fully qualified pathnames to prevent security problems.

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()/select() may be best mapping.
  - However poll()/select() 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()/select(). See sample1.c
  - 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).

UNIX user and group account mechanisms

- UNIX has a simple 32-bit (16 in the current Linux kernel) user id to represent any user. Likewise a 32/16 bit group id to represent a group. User and group number spaces are disjoint.
- "Foreign" users (from another machine) cannot be distinguished from a local user if their uid is the same.
  - YP/NIS and NIS+ are simply a way of getting a group of UNIX machines to agree on a common mapping for these numbers to user/group names.
- On-disk storage of file ownership is only the owning uid and gid.

Windows NT user and group account mechanisms (SIDs)

- NT uses a "SID" (security identifier) to store user and group identities. NT machines also have SIDs.
- SID number space is flat (users / groups / machines) all allocated from the same number space.
- Unlike uids, SIDs have a complex structure. A typical SID looks like :
  - S-1-5-21-<32 bits>-<32 bits>-<32 bits>-<32-bits>
  - The first '1' is the revision level of the SID.
  - The '5' is the "indentifying authority" (ie. who created it). 5 means an NT system (Samba uses this also).

NT SIDs (continued)

- The '21' is the sub-authority. 21 means accounts created by the Administrator (ie. not built in accounts). Well, mostly :-).
- The next 96 bits are a Domain or Machine ID.
  - All NT machines have a 96 bit unique identifier. This fact is significant and will be covered in a later slide.
- The final 32 bits are a "RID" (relative ID). This is the actual user or group ID within the 96-bit unique identifier.
- The overall SID design is unusably complex (and almost no programmers understand it).

Windows NT security model

- When a user logs onto an NT machine, their SID, as well as a list of the group SIDs they belong to, are stored in a kernel data structure known as an "access token".
  - An access token is associated with every process.
  - Threads share their owning processes token, unless they are impersonating a user.
  - This is identical to the process credentials in the Linux task_struct structure.
  - This group list is fetched from the account database (local account) or from the PDC/BDC (domain account).

UNIX security model

- UNIX has 'root' - all powerful, and "everyone else".
  - User logs on (username/password pair).
  - Login process (running as root) consults system databases, authenticates password and then determines the uid and list of gid's that the user belongs to.
  - Login process forks(), then sets this group list as the current group list onto the current process, sets the current process uid to the user uid in a one-way manor, then overlays the current process with the users logon shell.
  - No way for the user to get "root" privilege back.

UNIX security model (continued)

- No "magic" or hidden API's involved in this process.
- Any process running as root can do this (become another user).
- Authentication of the users password is optional for a root process.
  - Use getpwnam() to look up a users authentication structure. Compare with NetUserGetInfo on NT (getpwnam() is much easier to use).
  - root processes can get users hashed password (this is intentionally hidden on NT).

Win32 Impersonation

- Win32 threads can impersonate a connected user via a single API call.
- Under the covers this is done via the NTLM challenge/response mechanism.
- Such a mechanism is easily built into a UNIX protocol if needed (no need for plaintext passwords over the wire).
  - Layering the application protocol on top of a known crypto protocol (SSL or ssh) is an easier solution and is less likely to cause security breaches.
- Win32 SSPI is equivalent to the UNIX GSSAPI.

Config files vs the Windows registry

- Registry is a binary "data dump" for applications and the system.
  - Nicely designed API, however if it is corrupted the system is unusable.
- UNIX uses text files (usually in /etc) for critical system configuration.
- Server applications can place their config files in /etc, or within their own directory.
  - Very common on UNIX to have a 2-layer config mechanism, with system global config in a file in /etc, and user specific config in a dot file within the users home directory.
  - Similar to Win32 profiles but a cleaner mechanism.

Emulating the registry

- UNIX has libraries designed to store name/value pairs (where the value can be an arbitrary binary value just like the registry).
- libdb implements a b-tree/hash based storage mechanism used for such data.
  - Not transactional (although an Open Source implementation is).
  - No security on individual elements.
- See sample5.c for details on implementing a simple registry for string data. Easy to extend for binary also.

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 - more dragons

- Beware of Win32 code using the "ChangeNotification" API's.
  - These are kernel notifications of changes on the filesystem - UNIX does not support this.
- Be aware of file system differences, UNIX has no file "attributes" like SYSTEM or HIDDEN. To delete a file the directory must have write permission.
- Open files can be deleted on UNIX - useful for private application space (atomically removed on close).

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.

Application data portability

- Many Win32 programs just write "C" structures directly onto the wire or disk.
  - This would even break on Windows if a different compiler with different structure packing rules was used.
  - The Visual C++ monopoly (RIP Borland) means no-one cares.
- UNIX programs must take care to write out in a portable format.
  - No direct structure dumps.
  - Endian independent linearization.
  - Using xdr library can help greatly here.

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).