Cygwin User's Guide

DJ Delorie

Pierre Humblet

Geoffrey Noer

Cygwin User's Guide

by DJ Delorie, Pierre Humblet, and Geoffrey Noer

Revision History

Revision 0.0 1998-10-06Revised by: noer@cygnus.com
Initial revision
Revision 20.1.01999-02-08Revised by: Pierre.Humblet@eurecom.fr
Expand, describe Cygwin 20.1

Table of Contents

List of Examples

Chapter 1. Cygwin Overview

1.1. What is it?

The Cygwin tools are ports of the popular GNU development tools and utilities for Windows NT and 9x. They function through the use of the Cygwin library which provides the UNIX system calls and environment that these programs require.

With the tools installed, programmers may write Win32 console or GUI applications that make use of the standard Microsoft Win32 API and/or the Cygwin API. As a result, it is possible to easily port many significant UNIX programs without the need for extensive changes to the source code. This includes configuring and building most of the available GNU software (including the development tools included with the Cygwin distributions). Even if the compiler tools are of little to no use to you, you may have interest in the many standard UNIX utilities. They can be used both from the bash shell (provided) or from the command.com.

1.2. Are the Cygwin tools free software?

Yes. Parts are GNU software (gcc, gas, ld, etc...), parts are covered by the standard X11 license, some of it is public domain, some of it was written by Cygnus and placed under the GPL. None of it is shareware. You don’t have to pay anyone to use it but you should be sure to read the copyright section of the FAQ more more information on how the GNU General Public License may affect your use of these tools. If you intend to port a proprietary application using the Cygwin library, you may want the Cygwin proprietary-use license. For more information about the proprietary-use license, please contact sales@cygnus.com. Customers of the native Win32 GNUPro should feel free to submit bug reports and ask questions through the normal channels. All other questions should be sent to the project mailing list cygwin@sourceware.cygnus.com.

1.3. A brief history of the Cygwin project

The first thing done was to enhance the development tools (gcc, gdb, gas, et al) so that they could generate/interpret Win32 native object files.

The next task was to port the tools to Win NT/9x. We could have done this by rewriting large portions of the source to work within the context of the Win32 API. But this would have meant spending a huge amount of time on each and every tool. Instead, we took a substantially different approach by writing a shared library (the Cygwin DLL) that adds the necessary UNIX-like functionality missing from the Win32 API (fork, spawn, signals, select, sockets, etc.). We call this new interface the Cygwin API. Once written, it was possible to build working Win32 tools using UNIX-hosted cross-compilers, linking against this library.

From this point, we pursued the goal of producing native tools capable of rebuilding themselves under Windows 9x and NT (this is often called self-hosting). Since neither OS ships with standard UNIX user tools (fileutils, textutils, bash, etc...), we had to get the GNU equivalents working with the Cygwin API. Most of these tools were previously only built natively so we had to modify their configure scripts to be compatible with cross-compilation. Other than the configuration changes, very few source-level changes had to be made. Running bash with the development tools and user tools in place, Windows 9x and NT look like a flavor of UNIX from the perspective of the GNU configure mechanism. Self hosting was achieved as of the beta 17.1 release.

1.4. Expectations for UNIX Programmers

Developers coming from a UNIX background will find a set of utilities they are already comfortable using, including a working UNIX shell. The compiler tools are the standard GNU compilers most people will have previously used under UNIX, only ported to the Windows host. Programmers wishing to port UNIX software to Windows NT or 9x will find that the Cygwin library provides an easy way to port many UNIX packages, with only minimal source code changes.

1.5. Expectations for Windows Programmers

Developers coming from a Windows background will find a set of tools capable of writing console or GUI executables that rely on the Microsoft Win32 API. The linker and dlltool utility may be used to write Windows Dynamically Linked Libraries (DLLs). The resource compiler "windres" is also provided with the native Windows GNUPro tools. All tools may be used from the Microsoft command line prompt, with full support for normal Windows pathnames.

1.6. Highlights of Cygwin Functionality

1.6.1. Introduction

When a binary linked against the library is executed, the Cygwin DLL is loaded into the application’s text segment. Because we are trying to emulate a UNIX kernel which needs access to all processes running under it, the first Cygwin DLL to run creates shared memory areas that other processes using separate instances of the DLL can access. This is used to keep track of open file descriptors and assist fork and exec, among other purposes. In addition to the shared memory regions, every process also has a per_process structure that contains information such as process id, user id, signal masks, and other similar process-specific information.

The DLL is implemented using the Win32 API, which allows it to run on all Win32 hosts. Because processes run under the standard Win32 subsystem, they can access both the UNIX compatibility calls provided by Cygwin as well as any of the Win32 API calls. This gives the programmer complete flexibility in designing the structure of their program in terms of the APIs used. For example, they could write a Win32-specific GUI using Win32 API calls on top of a UNIX back-end that uses Cygwin.

Early on in the development process, we made the important design decision that it would not be necessary to strictly adhere to existing UNIX standards like POSIX.1 if it was not possible or if it would significantly diminish the usability of the tools on the

Win32 platform. In many cases, an environment variable can be set to override the default behavior and force standards compliance.

1.6.2. Supporting both Windows NT and 9x

While Windows 95 and Windows 98 are similar enough to each other that we can safely ignore the distinction when implementing Cygwin, Windows NT is an extremely different operating system. For this reason, whenever the DLL is loaded, the library checks which operating system is active so that it can act accordingly.

In some cases, the Win32 API is only different for historical reasons. In this situation, the same basic functionality is available under Windows 9x and NT but the method used to gain this functionality differs. A trivial example: in our implementation of uname, the library examines the sysinfo.dwProcessorType structure member to figure out the processor type under Windows 9x. This field is not supported in NT, which has its own operating system-specific structure member called sysinfo.wProcessorLevel.

Other differences between NT and 9x are much more fundamental in nature. The best example is that only NT provides a security model.

1.6.3. Permissions and Security

Windows NT includes a sophisticated security model based on Access Control Lists (ACLs). Although some modern UNIX operating systems include support for ACLs, Cygwin maps Win32 file ownership and permissions to the more standard, older UNIX model. The chmod call maps UNIX-style permissions back to the Win32 equivalents. Because many programs expect to be able to find the /etc/passwd and /etc/group files, we provide utilities that can be used to construct them from the user and group information provided by the operating system.

Under Windows NT, the administrator is permitted to chown files. There is currently no mechanism to support the setuid concept or API call. Although we hope to support this functionality at some point in the future, in practice, the programs we have ported have not needed it.

Under Windows 9x, the situation is considerably different. Since a security model is not provided, Cygwin fakes file ownership by making all files look like they are owned by a default user and group id. As under NT, file permissions can still be determined by examining their read/write/execute status. Rather than return an unimplemented error, under Windows 9x, the chown call succeeds immediately without actually performing any action whatsoever. This is appropriate since essentially all users jointly own the files when no concept of file ownership exists.

It is important that we discuss the implications of our "kernel" using shared memory areas to store information about Cygwin processes. Because these areas are not yet protected in any way, in principle a malicious user could modify them to cause unexpected behavior in Cygwin processes. While this is not a new problem under Windows 9x (because of the lack of operating system security), it does constitute a security hole under Windows NT. This is because one user could affect the Cygwin programs run by another user by changing the shared memory information in ways that they could not in a more typical WinNT program. For this reason, it is not appropriate to use Cygwin in high-security applications. In practice, this will not be a major problem for most uses of the library.

1.6.4. File Access

Cygwin supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. Paths coming into the DLL are translated from Win32 to POSIX as needed. As a result, the library believes that the file system is a POSIX-compliant one, translating paths back to Win32 paths whenever it calls a Win32 API function. UNC pathnames (starting with two slashes) are supported.

The layout of this POSIX view of the Windows file system space is stored in the Windows registry. While the slash (’/’) directory points to the system partition by default, this is easy to change with the Cygwin mount utility. In addition to selecting the slash partition, it allows mounting arbitrary Win32 paths into the POSIX file system space. Many people use the utility to mount each drive letter under the slash partition (e.g. C:\ to /c, D:\ to /d, etc...).

The library exports several Cygwin-specific functions that can be used by external

programs to convert a path or path list from Win32 to POSIX or vice versa. Shell scripts and Makefiles cannot call these functions directly. Instead, they can do the same path translations by executing the cygpath utility program that we provide with Cygwin.

Win32 file systems are case preserving but case insensitive. Cygwin does not currently support case distinction because, in practice, few UNIX programs actually rely on it. While we could mangle file names to support case distinction, this would add unnecessary overhead to the library and make it more difficult for non-Cygwin applications to access those files.

Symbolic links are emulated by files containing a magic cookie followed by the path to which the link points. They are marked with the System attribute so that only files with that attribute have to be read to determine whether or not the file is a symbolic link. Hard links are fully supported under Windows NT on NTFS file systems. On a FAT file system, the call falls back to simply copying the file, a strategy that works in many cases.

The inode number for a file is calculated by hashing its full Win32 path. The inode number generated by the stat call always matches the one returned in d_ino of the dirent structure. It is worth noting that the number produced by this method is not guaranteed to be unique. However, we have not found this to be a significant problem because of the low probability of generating a duplicate inode number.

1.6.5. Text Mode vs. Binary Mode

Interoperability with other Win32 programs such as text editors was critical to the success of the port of the development tools. Most Cygnus customers upgrading from the older DOS-hosted toolchains expected the new Win32-hosted ones to continue to work with their old development sources.

Unfortunately, UNIX and Win32 use different end-of-line terminators in text files. Consequently, carriage-return newlines have to be translated on the fly by Cygwin into a single newline when reading in text mode. The control-z character is interpreted as a valid end-of-file character for a similar reason.

This solution addresses the compatibility requirement at the expense of violating the POSIX standard that states that text and binary mode will be identical. Consequently, processes that attempt to lseek through text files can no longer rely on the number of bytes read as an accurate indicator of position in the file. For this reason, the CYGWIN environment variable can be set to override this behavior.

1.6.6. ANSI C Library

We chose to include Cygnus' own existing ANSI C library "newlib" as part of the library, rather than write all of the lib C and math calls from scratch. Newlib is a BSD-derived ANSI C library, previously only used by cross-compilers for embedded systems development.

The reuse of existing free implementations of such things as the glob, regexp, and getopt libraries saved us considerable effort. In addition, Cygwin uses Doug Lea's free malloc implementation that successfully balances speed and compactness. The library accesses the malloc calls via an exported function pointer. This makes it possible for a Cygwin process to provide its own malloc if it so desires.

1.6.7. Process Creation

The fork call in Cygwin is particularly interesting because it does not map well on top of the Win32 API. This makes it very difficult to implement correctly. Currently, the Cygwin fork is a non-copy-on-write implementation similar to what was present in early flavors of UNIX.

The first thing that happens when a parent process forks a child process is that the parent initializes a space in the Cygwin process table for the child. It then creates a suspended child process using the Win32 CreateProcess call. Next, the parent process calls setjmp to save its own context and sets a pointer to this in a Cygwin shared memory area (shared among all Cygwin tasks). It then fills in the child's .data and .bss sections by copying from its own address space into the suspended child's address space. After the child's address space is initialized, the child is run while the parent

waits on a mutex. The child discovers it has been forked and longjumps using the saved jump buffer. The child then sets the mutex the parent is waiting on and blocks on another mutex. This is the signal for the parent to copy its stack and heap into the child, after which it releases the mutex the child is waiting on and returns from the fork call. Finally, the child wakes from blocking on the last mutex, recreates any memory-mapped areas passed to it via the shared area, and returns from fork itself.

While we have some ideas as to how to speed up our fork implementation by reducing the number of context switches between the parent and child process, fork will almost certainly always be inefficient under Win32. Fortunately, in most circumstances the spawn family of calls provided by Cygwin can be substituted for a fork/exec pair with only a little effort. These calls map cleanly on top of the Win32 API. As a result, they are much more efficient. Changing the compiler's driver program to call spawn instead of fork was a trivial change and increased compilation speeds by twenty to thirty percent in our tests.

However, spawn and exec present their own set of difficulties. Because there is no way to do an actual exec under Win32, Cygwin has to invent its own Process IDs (PIDs). As a result, when a process performs multiple exec calls, there will be multiple Windows PIDs associated with a single Cygwin PID. In some cases, stubs of each of these Win32 processes may linger, waiting for their exec'd Cygwin process to exit.

1.6.8. Signals

When a Cygwin process starts, the library starts a secondary thread for use in signal handling. This thread waits for Windows events used to pass signals to the process. When a process notices it has a signal, it scans its signal bitmask and handles the signal in the appropriate fashion.

Several complications in the implementation arise from the fact that the signal handler operates in the same address space as the executing program. The immediate consequence is that Cygwin system functions are interruptible unless special care is taken to avoid this. We go to some lengths to prevent the sig_send function that sends signals from being interrupted. In the case of a process sending a signal to another process, we place a mutex around sig_send such that sig_send will not be interrupted

until it has completely finished sending the signal.

In the case of a process sending itself a signal, we use a separate semaphore/event pair instead of the mutex. sig_send starts by resetting the event and incrementing the semaphore that flags the signal handler to process the signal. After the signal is processed, the signal handler signals the event that it is done. This process keeps intraprocess signals synchronous, as required by POSIX.

Most standard UNIX signals are provided. Job control works as expected in shells that support it.

1.6.9. Sockets

Socket-related calls in Cygwin simply call the functions by the same name in Winsock, Microsoft's implementation of Berkeley sockets. Only a few changes were needed to match the expected UNIX semantics - one of the most troublesome differences was that Winsock must be initialized before the first socket function is called. As a result, Cygwin has to perform this initialization when appropriate. In order to support sockets across fork calls, child processes initialize Winsock if any inherited file descriptor is a socket.

Unfortunately, implicitly loading DLLs at process startup is usually a slow affair. Because many processes do not use sockets, Cygwin explicitly loads the Winsock DLL the first time it calls the Winsock initialization routine. This single change sped up GNU configure times by thirty percent.

1.6.10. Select

The UNIX select function is another call that does not map cleanly on top of the Win32 API. Much to our dismay, we discovered that the Win32 select in Winsock only worked on socket handles. Our implementation allows select to function normally when given different types of file descriptors (sockets, pipes, handles, and a custom /dev/windows Windows messages pseudo-device).

Upon entry into the select function, the first operation is to sort the file descriptors into the different types. There are then two cases to consider. The simple case is when at least one file descriptor is a type that is always known to be ready (such as a disk file). In that case, select returns immediately as soon as it has polled each of the other types to see if they are ready. The more complex case involves waiting for socket or pipe file descriptors to be ready. This is accomplished by the main thread suspending itself, after starting one thread for each type of file descriptor present. Each thread polls the file descriptors of its respective type with the appropriate Win32 API call. As soon as a thread identifies a ready descriptor, that thread signals the main thread to wake up. This case is now the same as the first one since we know at least one descriptor is ready. So select returns, after polling all of the file descriptors one last time.

Chapter 2. Setting Up Cygwin

2.1. Cygwin Contents

The following packages are included in the full release:

Development tools: binutils, bison, byacc, dejagnu, diff, expect, flex, gas, gcc, gdb, itcl, ld, libstdc++, make, patch, tcl, tix, tk

User tools: ash, bash, bzip2, diff, fileutils, findutils, gawk, grep, gzip, less, m4, sed, shellutils, tar, textutils, time

The user tools release only contains the user tools.

Full source code is available for these tools.

2.2. Installing the binary release

Important! Be sure to remove any older versions of the Cygwin tools from your PATH environment variable so you do not execute them by mistake.

Connect to one of the ftp servers listed in http://sourceware.cygnus.com/cygwin/mirrors.html and **cd** to the directory containing the latest release.

If you want the development tools and the programs necessary to run the GNU configure mechanism, you should download the full binary release called `full.exe`. If you only care about the user tools listed above, download `usertools.exe` instead.

If you have an unreliable connection, download the appropriate binary in smaller chunks instead. For the split full installer, get the files in the ‘full-split’ subdirectory. Once downloaded, combine the split files at the command prompt by doing a:

```
C:\Cygnus\>copy /b xaa + xab + xac + ... + xak + xal full.exe
C:\Cygnus\>del xa*.*
```

A similar process can be used for the user tools.

Once you have installed the executable on your system, run it. First off, the installer will prompt you for a location to extract the temporary files it needs to install the release on your system. The default should be fine for most people.

Next it will ask you to choose an install location. The default is *`system-drive`*`:\cygnus\cygwin-b20`. Feel free to choose another location if you would prefer.

Finally, it will ask you for the name of the Program Files folder shortcut to add. By default, the installer will create a ‘Cygwin B20’ entry in a folder called ‘Cygnus Solutions’. When this step is completed, it will install the tools and exit.

If you should ever want to uninstall the tools, you may do so via the "Add/Remove Programs" control panel.

At this point you should be able to look under the start menu and select "Cygwin Beta 20" (or whatever you named it). This will pop up a bash shell with special environment variables set up for you. If you are running Windows 95 or 98 and are faced with the error message "Out of environment space", you need to increase the amount of environment space. Adding the line **shell=C:\command.com /e:4096 /p** to the file `C:\CONFIG.SYS` and then rebooting should do the trick if `C:` is your system drive letter.

If you want to install the sources follow the instructions in the next section, else go directly to Section 2.4 to complete your system setup.

2.3. Installing the source code

Before downloading the source code corresponding to the release, you should install the latest release of the tools (either the full release or just the user tools).

Create the directory that will house the source code. **cd** there.

Connect to one of the ftp servers listed above and **cd** to the directory containing the latest release.

The source code is split into two units: user tools and development tools. If you want the user tools source code, **cd** into the `user-src-split` subdirectory. Download the files there. If you want the development tools sources, **cd** into the `dev-src-split` subdirectory. Download the files there.

Back in the Windows command shell, for the user tools source:

```
C:\Cygnus\> copy /b xba + xbb + xbc + xbd + xbe + xbf + xbg user-
src.tar.bz2
C:\Cygnus\> del xb*.*
C:\Cygnus\> bunzip2 user-src.tar.bz2
C:\Cygnus\> tar xvf user-src.tar
```

For the development tools source:

```
C:\Cygnus\> copy /b xca + xcb + xcc + xcd + ... + xck + xcl dev-
src.tar.bz2
C:\Cygnus\> del xc*.*
C:\Cygnus\> bunzip2 dev-src.tar.bz2
C:\Cygnus\> tar xvf dev-src.tar
```

Both will expand into a directory called `src`.

Note: if you want the sources corresponding to everything in the full.exe binary installer, you will need to download and expand both the `user-src.tar.bz2` and `dev-src.tar.bz2` source archives!

2.4. Directory Structure

Cygwin knows how to emulate a standard UNIX directory structure, to some extent. This is done through the use of mount tables that map Win32 paths to POSIX ones. The mount table may be set up and modified with the **mount** command. This section explains how to properly organize the structure.

When you set up the system you should decide where you want the root to be mapped. Possible choices are the root of your Windows system, such as `c:` or a directory such as `c:\progra~1\root`.

Execute the following commands inside bash as it is difficult to change the position of the root from the Windows command prompt. Changing the mount points may invalidate PATH, if this happens simply exit and relaunch bash. Create the directory if needed, then **umount /** the current root and **mount** it in its new place. You also have to decide if you want to use text or binary mode.

Next, create the traditional main UNIX directories, with the following command (in some shells it is necessary to issue separate **mkdir** commands, each with a single argument).

```
/$ mkdir /tmp /bin /etc /var /usr
```

Next we will initialize the content of these directories.

You should make sure that you always have a valid `/tmp` directory. If you want to avoid creating a real `/tmp`, you can use the **mount** utility to point `/tmp` to another directory, such as `c:\tmp`, or create a symbolic link `/tmp` to point to such a directory.

The `/bin` directory should contain the shell `sh.exe`. You have three choices. The first is to copy this program from the Cygnus `bin` directory. The second is to use **mount** to mount the Cygnus `bin` directory to `/bin` (the advantage of this approach is that your PATH will be shorter inside bash). The third is to make `/bin` a symbolic link to the Cygnus `bin` directory.

Note that Cygwin comes with two shells: **bash.exe** and **sh.exe**, which is based on **ash**. The system is faster when **ash** is used as the non-interactive shell. The only functionality supported in **ash** is that of the traditional **sh**. In case of trouble with **ash** make **sh.exe** point to **bash.exe**.

We now turn to `/etc`. You may want to copy in it the `termcap` file from the Cygnus `etc` directory, although the defaults built into the programs suffice for the normal console. You may also use **mount** or create as symbolic link to the Cygnus `etc`, just as for `/bin` above.

Under Windows NT, if you want to create `/etc/passwd` and `/etc/group` (i.e. so that **whoami** works and **ls -l** replaces the UID with a name) just do this:

```
/$ cd /etc
/etc$ mkpasswd -l > /etc/passwd
/etc$ mkgroup -l > /etc/group
```

Future changes to your NT registry will NOT be reflected in `/etc/passwd` or `/etc/group` after this so you may want to regenerate these files periodically. Under Windows 9x, you can create and edit these files with a text editor.

The **who** command requires the `/var/run/utmp` to exist. Create it if you wish. The system also logs information in `/var/log/wtmp`, if it exists.

The `/usr` directory is not used by the Cygwin system but it is a standard place to install optional packages.

You may also want to mount directories such as `/a` and `/d` to refer to your local and network drives.

You do not need to create `/dev` in order to set up mounts for devices such as `/dev/null` as these are already automatically simulated inside the Cygwin library.

2.5. Environment Variables

Before starting bash, you must set some environment variables, some of which can also be set or modified inside bash. Cygnus provides you with a .bat file where the most important ones are set before bash in launched. This is the safest way to launch bash initially. The .bat file is installed by default in `\cygnus\cygwin-b20/cygnus.bat` and pointed to in the Start Menu. You can edit it to your liking.

The CYGWIN variable is used to configure many global settings for the Cygwin runtime system. Initially you can leave CYGWIN unset or set it to `tty` (e.g. to support job control with ^Z etc...) using a syntax like this in the DOS shell, before launching bash.

```
C:\Cygnus\> set CYGWIN=tty notitle strace=0x1
```

The PATH environment variable is used by Cygwin applications as a list of directories to search for executable files to run. This environment variable is converted from Windows format (e.g. `C:\WinNT\system32;C:\WinNT`) to UNIX format (e.g., `/WinNT/system32:/WinNT`) when a Cygwin process first starts. Set it so that it contains at least the Cygnus `bin` directory `C:\cygnus\cygwin-b20\H-i586-cygwin32\bin` before launching bash.

The HOME environment variable is used by many programs to determine the location of your home directory and we recommend that it be defined. This environment variable is also converted from Windows format when a Cygwin process first starts. Set it to point to your home directory before launching bash.

make uses an environment variable MAKE_MODE to decide if it uses `command.com` or `/bin/sh` to run command lines. If you are getting strange errors from **make** about "/c not found", set MAKE_MODE to `UNIX` at the command prompt or in bash.

```
C:\Cygnus\> set MAKE_MODE=UNIX
```

```
/Cygnus$ export MAKE_MODE=UNIX
```

The TERM environment variable specifies your terminal type. You can set it to `cygwin`.

The LD_LIBRARY_PATH environment variable is used by the Cygwin function `dlopen ()` as a list of directories to search for .dll files to load. This environment variable is converted from Windows format to UNIX format when a Cygwin process first starts. Most Cygwin applications do not make use of the `dlopen ()` call and do not need this variable.

2.6. NT security and the `ntsec` usage

The design goal of the ntsec patch was to get a more UNIX like permission structure based upon the security features of Windows NT. To describe the changes, I will give a

short overview of NT security in chapter one.

Chapter two discusses the changes in ntsec related to privileges on processes.

Chapter three shows the basics of UNIX like setting of file permissions.

Chapter four talks about the advanced settings introduced in release 1.1

Chapter five illustrates the permission mapping leak of Windows NT.

Chapter six describes in short the new acl API since release 1.1

The setting of UNIX like object permissions is controlled by the new CYGWIN variable setting `(no)ntsec`. On NT ntsec is now turned on by default.

2.6.1. NT security

The NT security allows a process to allow or deny access of different kind to ‘objects’. ‘Objects’ are files, processes, threads, semaphores, etc.

The main data structure of NT security is the ‘security descriptor’ (SD) structure. It explains the permissions, that are granted (or denied) to an object and contains information, that is related to so called ‘security identifiers’ (SID).

A SID is a unique identifier for users, groups and domains. SIDs are comparable to UNIX UIDs and GIDs, but are more complicated because they are unique across networks. Example:

SID of a system ‘foo’:

```
S-1-5-21-165875785-1005667432-441284377
```

SID of a user ‘johndoe’ of the system ‘foo’:

```
S-1-5-21-165875785-1005667432-441284377-1023
```

The above example shows the convention for printing SIDs. The leading ‘S’ should show that it is a SID. The next number is a version number which is always 1. The next number is the so called ‘top-level authority’ that identifies the source that issued the SID.

While each system in a NT network has it's own SID, the situation is modified in NT domains: The SID of the domain controller is the base SID for each domain user. If an NT user has one account as domain user and another account on his local machine, this accounts are under any circumstances DIFFERENT, regardless of the usage of the same user name and password!

SID of a domain 'bar':

```
S-1-5-21-186985262-1144665072-740312968
```

SID of a user 'johndoe' in the domain 'bar':

```
S-1-5-21-186985262-1144665072-740312968-1207
```

The last part of the SID, the so called 'relative identifier' (RID), is by default used as UID and/or GID under cygwin. As the name and the above example implies, this id is unique only relative to one system or domain.

Note, that it's possible, that an user has the same RID on two different systems. The resulting SIDs are nevertheless different, so the SIDs are representing different users in an NT network.

There is a big difference between UNIX IDs and NT SIDs, the existence of the so called 'well known groups'. For example UNIX has no GID for the group of 'all users'. NT has an SID for them, called 'Everyone' in the English versions. The SIDs of well-known groups are not unique across an NT network but their meanings are unmistakable. Examples of well-known groups:

```
everyone                        S-1-1-0
creator/owner                   S-1-3-0
batch process (via 'at')        S-1-5-3
authenticated users             S-1-5-11
system                          S-1-5-18
```

The last important group of SIDs are the 'predefined groups'. This groups are used mainly on systems outside of domains to simplify the administration of user permissions. The corresponding SIDs are not unique across the network so they are interpreted only locally:

```
administrators                  S-1-5-32-544
users                           S-1-5-32-545
guests                          S-1-5-32-546
...
```

Now, how are permissions given to objects? A process may assign an SD to the object. The SD of an object consists of three parts:

- the SID of the owner
- the SID of the group
- a list of SIDs with their permissions, called ‘access control list’ (ACL)

UNIX is able to create three different permissions, the permissions for the owner, for the group and for the world. In contrast the ACL has a potentially infinite number of members. Every member is a so called ‘access control element’ (ACE). An ACE contains three parts:

- the type of the ACE
- permissions, described with a DWORD
- the SID, for which the above mentioned permissions are set

The two important types of ACEs are the ‘access allowed ACE’ and the ‘access denied ACE’. The ntsec patch only uses ‘access allowed ACEs’.

The possible permissions on objects are more complicated than in UNIX. For example, the permission to delete an object is different from the write permission.

With the aforementioned method NT is able to grant or revoke permissions to objects in a far more specific way. But what about cygwin? In a POSIX environment it would be fine to have the security behavior of a POSIX system. The NT security model is MOSTLY able to reproduce the POSIX model. The ntsec patch tries to do this in cygwin.

You ask "Mostly? Why mostly???" Because there’s a leak in the NT model. I will describe that in detail in chapter 4.

The creation of explicit object security is a bit complicated, so typically only two simple variations are used:

- default permissions, computed by the operating system
- each permission to everyone

For parameters to functions that create or open securable objects another data structure is used, the ‘security attributes’ (SA). This structure contains an SD and a flag, that specifies whether the returned handle to the created or opened object is inherited to child processes or not. This property is not important for the ntsec patch description, so in this document SDs and SAs are more or less identical.

2.6.2. Process privileges

Any process started under control of cygwin has a semaphore attached to it, that is used for signaling purposes. The creation of this semaphore can be found in sigproc.cc, function ‘getsem’. The first parameter to the function call ‘CreateSemaphore’ is an SA. Without ntsec patch this SA assigns default security to the semaphore. There is a simple disadvantage: Only the owner of the process may send signals to it. Or, in other words, if the owner of the process is not a member of the administrators’ group, no administrator may kill the process! This is especially annoying, if processes are started via service manager.

The ntsec patch now assigns an SA to the process control semaphore, that has each permission set for the user of the process, for the administrators’ group and for ‘system’, which is a synonym for the operating system itself. The creation of this SA is done by the function ‘sec_user’, that can be found in ‘shared.cc’. Each member of the administrators’ group is now allowed to send signals to any process created in cygwin, regardless of the process owner.

Moreover, each process now has the appropriate security settings, when it is started via ‘CreateProcess’. You will find this in function ‘spawn_guts’ in module ‘spawn.cc’. The security settings for starting a process in another user context have to add the sid of the new user, too. In the case of the ‘CreateProcessAsUser’ call, sec_user creates an SA with an additional entry for the sid of the new user.

2.6.3. File permissions

If ntsec is turned on, file permissions are set as in UNIX. An SD is assigned to the file containing the owner and group and ACEs for the owner, the group and ‘Everyone’.

The complete settings of UNIX like permissions can be found in the file ‘security.cc’. The two functions ‘get_nt_attribute’ and ‘set_nt_attribute’ are the main code. The reading and writing of the SDs is done by the functions ‘read_sd’ and ‘write_sd’. ‘write_sd’ uses the function ‘BackupRead’ instead of the simpler function ‘SetFileSecurity’ because the latter is unable to set owners different from the caller.

If you are creating a file ‘foo’ outside of cygwin, you will see something like the following on **ls -ln**:

If your login is member of the administrators’ group:

```
rwxrwxrwx 1  544  513  ... foo
```

if not:

```
rwxrwxrwx 1  1000  513  ... foo
```

Note the user and group IDs. 544 is the UID of the administrators’ group. This is a ‘feature’ `:-P` of WinNT. If one is a member of the administrators’ group, every file, that he has created is owned by the administrators’ group, instead by him.

The second example shows the UID of the first user, that has been created with NT’s the user administration tool. The users and groups are sequentially numbered, starting with 1000. Users and groups are using the same numbering scheme, so a user and a group don’t share the same ID.

In both examples the GID 513 is of special interest. This GID is a well known group with different naming in local systems and domains. Outside of domains the group is named ’None’ (‘Kein’ in German, ‘Aucun’ in French, etc.), in domains it is named ’Domain Users’. Unfortunately, the group ‘None’ is never shown in the user admin tool outside of domains! This is very confusing but it seems that this has no negativ influences.

To work correctly the ntsec patch depends on reasoned files `/etc/passwd/` and `/etc/group`. In cygwin release 1.0 the names and the IDs must correspond to the appropriate NT IDs! The IDs used in cygwin are the RID of the NT SID, as mentioned earlier. An SID of e.g. the user ‘corinna’ on my NT workstation:

```
S-1-5-21-165875785-1005667432-441284377-1000
```

Note the last number: It’s the RID 1000, the cygwin’s UID.

Unfortunately, workstations and servers outside of domains are not able to set primary groups! In these cases, where there is no correlation of users to primary groups, NT returns 513 (None) as primary group, regardless of the membership to existing local groups.

when using **mkpasswd -l -g** on such systems, you have to change the primary group by hand if ‘None’ as primary group is not what you want (and I’m sure, it’s not what you want!)

To get help in creating correct passwd and group files, look at the following examples, that are part of my files. With the exception of my personal user entry, all entries are well known entries. For a better understanding, the names are translated to the equivalents of the English NT version.

Example 2-1. /etc/passwd

```
everyone:*:0:0:::
system:*:18:18:::
administrator::500:544::/home/root:/bin/bash
guest:*:501:546:::
administrators:*:544:544::/home/root:
corinna::1000:547:Corinna Vinschen:/home/corinna:/bin/tcsh
```

Example 2-2. /etc/group

```
everyone::0:
system::18:
none::513:
```

```
administrators::544:
users::545:
guests::546:
powerusers::547:
```

Groups may be mentioned in the passwd file, too. This has two advantages:

- Because NT assigns them to files as owners, a **ls -l** is often better readable.
- Moreover it's possible to assigned them to files as owners with cygwin's **chown**.

The group 'system' is the aforementioned synonym for the operating system itself and is normally the owner of processes, that are started through service manager. The same is true for files, that are created by processes, which are started through service manager.

2.6.4. New since Cygwin release 1.1

In Cygwin release 1.1 a new technique of using the `/etc/passwd` and `/etc/group` is introduced.

Both files may now contain SIDs of users and groups. They are saved in the last field of pw_gecos in `/etc/passwd` and in the gr_passwd field in `/etc/group`.

This has the following advantages:

- ntsec works better in domain environments.
- Accounts (users and groups) may get another name in cygwin that their NT account name. The name in `/etc/passwd` or `/etc/group` is transparently used by cygwin applications (eg. **chown**, **chmod**, **ls**):

  `root::500:513::/home/root:/bin/sh`

  instead of

  `adminstrator::500:513::/home/root:/bin/sh`

Caution: If you like to use the account as login account via **telnet** etc. you have to remain the name unchanged or you have to use a special version of **login** which will be part of the release 1.1 soon.

- Cygwin UIDs and GIDs are now not necessarily the RID part of the NT SID:

  ```
  root::0:513:S-1-5-21-54355234-56236534-345635656-
  500:/home/root:/bin/sh
  ```

  instead of

  ```
  root::500:513::/home/root:/bin/sh
  ```

- As in U*X systems UIDs and GIDs numbering scheme now don't influence each other. So it's possible to have same Id's for a user and a group:

  Example 2-3. /etc/passwd:

  ```
  root::0:0:S-1-5-21-54355234-56236534-345635656-
  500:/home/root:/bin/sh
  ```

  Example 2-4. /etc/group:

  ```
  root:S-1-5-32-544:0:
  ```

The tools **mkpasswd** and **mkgroup** create the needed entries by default. If you don't want that you can use the options `-s` or `-no-sids`. In this case ntsec behaves like the previous version.

Please note that the pw_gecos field in `/etc/passwd` is defined as a comma seperated list. The SID has to be the last field!

As aforementioned you are able to use cygwin account names different from the NT account names. If you want to login thru 'telnet' or something else you have to use the special **login**. You may then add another field to pw_gecos which contains the NT user name including it's domain. So you are able to login as each domain user. The syntax is easy: Just add an entry of the form U-ntdomain\ntusername to the pw_gecos field. Note that the SID must still remain the last field in pw_gecos!

```
the_king::1:1:Elvis Presley,U-STILLHERE\elvis,S-1-5-21-1234-
5678-9012-1000:/bin/sh
```

For a local user just drop the domain:

```
the_king::1:1:Elvis Presley,U-elvis,S-1-5-21-1234-5678-9012-
1000:/bin/sh
```

In each case the password of the user is taken from the NT user database, NOT from the passwd file!

2.6.5. The mapping leak

Now its time to point out the leak in the NT permissions. The official documentation explains in short the following:

- access allow ACEs are accumulated regarding to the group membership of the caller.
- The order of ACEs is important. The system reads them in sequence until either any needed right is denied or all needed rights are granted. Later ACEs are then not taken into account.
- ALl access denied ACEs _should_ precede any access allowed ACE.

Note that the last rule is a preference, not a law. NT will correctly deal with the ACL regardless of the sequence order. The second rule is not modified to get the ACEs in the prefered order.

Unfortunately the security tab of the NT4 explorer is completely unable to deal with access denied ACEs while the explorer of W2K rearranges the order of the ACEs before you can read them. Thank God, the sort order remains unchanged if one presses the Cancel button.

You still ask "Where is the leak?" NT ACLs are unable to reflect each possible combination of POSIX permissions. Example:

```
rw-r-xrw-
```

1st try:

```
UserAllow:   110
GroupAllow:  101
OthersAllow: 110
```

Hmm, because of the accumulation of allow rights the user may execute because the group may execute.

2st try:

```
UserDeny:    001
GroupAllow:  101
OthersAllow: 110
```

Now the user may read and write but not execute. Better? No! Unfortunately the group may write now because others may write.

3rd try:

```
UserDeny:    001
GroupDeny:   010
GroupAllow:  001
OthersAllow: 110
```

Now the group may not write as intended but unfortunately the user may not write anymore, too. How should this problem be solved? According to the official rules a UserAllow has to follow the GroupDeny but it’s easy to see that this can never be solved that way.

The only chance:

```
UserDeny:    001
UserAllow:   010
GroupDeny:   010
GroupAllow:  001
OthersAllow: 110
```

Again: This works for both, NT4 and W2K. Only the GUIs aren't able to deal with that order.

2.6.6. New acl API

For dealing with ACLs Cygwin now has the acl API as it's implemented in newer versions of Solaris. The new data structure for a single ACL entry (ACE in NT terminology) is defined in `sys/acl.h` as:

```
typedef struct acl {
  int     a_type;  /* entry type */
  uid_t   a_id;    /* UID | GID */
  mode_t  a_perm;  /* permissions */
} aclent_t;
```

The a_perm member of the aclent_t type contains only the bits for read, write and execute as in the file mode. If eg. read permission is granted, all read bits (S_IRUSR, S_IRGRP, S_IROTH) are set. CLASS_OBJ or MASK ACL entries are not fully implemented yet.

The new API calls are

```
acl(2), facl(2)
aclcheck(3),
aclsort(3),
acltomode(3), aclfrommode(3),
acltopbits(3), aclfrompbits(3),
acltotext(3), aclfromtext(3)
```

Like in Solaris, Cygwin has two new commands for working with ACLs on the command line: **getfacl** and **setfacl**.

Online man pages for the aforementioned commands and API calls can be found on eg. http://docs.sun.com

2.7. Customizing bash

To set bash up so that cut and paste work properly, click on the "Properties" button of the window, then on the "Misc" tab. Make sure that "Quick Edit" is checked and "Fast Pasting" isn't. These settings will be remembered next time you run bash from that shortcut. Similarly you can set the working directory inside the "Program" tab. The entry "%HOME%" is valid.

Your home directory should contain three initialization files that control the behavior of bash. They are `.profile`, `.bashrc` and `.inputrc`. These initialization files will only be read if HOME is defined before starting bash.

`.profile` (other names are also valid, see the bash man page) contains bash commands. It is executed when bash is started as login shell, e.g. from the command **bash –login** (the provided .bat file does not set the switch). This is a useful place to define and export environment variables and bash functions that will be used by bash and the programs invoked by bash. It is a good place to redefine PATH if needed. We recommend adding a ":." to the end of PATH to also search the current working directory (contrary to DOS, the local directory is not searched by default). Also to avoid delays you should either **unset** MAILCHECK or define MAILPATH to point to your existing mail inbox.

`.bashrc` is similar to `.profile` but is executed each time an interactive bash shell is launched. It serves to define elements that are not inherited through the environment, such as aliases. If you do not use login shells, you may want to put the contents of `.profile` as discussed above in this file instead.

```
shopt -s nocaseglob
```

will allow bash to glob filenames in a case-insensitive manner. Note that `.bashrc` is not called automatically for login shells. You can source it from `.profile`.

`.inputrc` controls how programs using the readline library (including bash) behave. It is loaded automatically. The full details are in the `readline.info`. Due to a bug in the current readline version, `.inputrc` cannot contain \r, even on text mounted systems. Consider the following settings:

```
# Make Bash 8bit clean
set meta-flag on
set convert-meta off
set output-meta on
# Ignore case while completing
set completion-ignore-case on
```

The first three commands allow bash to display 8-bit characters, useful for languages with accented characters. The last line makes filename completion case insensitive, which can be convenient in a Windows environment.

Chapter 3. Using Cygwin

This chapter explains some key differences between the Cygwin environment and traditional UNIX systems. It assumes a working knowledge of standard UNIX commands.

3.1. Mapping path names

3.1.1. Introduction

Cygwin supports both Win32- and POSIX-style paths, where directory delimiters may be either forward or back slashes. UNC pathnames (starting with two slashes and a network name) are also supported.

POSIX operating systems (such as Linux) do not have the concept of drive letters. Instead, all absolute paths begin with a slash (instead of a drive letter such as "c:") and all file systems appear as subdirectories (for example, you might buy a new disk and make it be the `/disk2` directory).

Because many programs written to run on UNIX systems assume the existance of a single unified POSIX file system structure, Cygwin maintains a special internal POSIX view of the Win32 file system that allows these programs to successfully run under Windows. Cygwin uses this mapping to translate between Win32 and POSIX paths as necessary.

3.1.2. The Cygwin Mount Table

The **mount** utility program is used to to map Win32 drives and network shares into Cygwin’s internal POSIX directory tree. This is a similar concept to the typical UNIX mount program. For those people coming from a Windows background, the **mount** utility is very similar to the old DOS **join**, in that it makes your drive letters appear as

subdirectories somewhere else.

The mapping is stored in the current user's Cygwin *mount table* in the Windows registry so that the information will be retrieved next time the user logs in. Because it is sometimes desirable to have system-wide as well as user-specific mounts, there is also a system-wide mount table that all Cygwin users inherit. The system-wide table may only be modified by a user with the appropriate priviledges (Administrator priviledges in Windows NT).

The current user's table is located under "HKEY_CURRENT_USER/Software/Cygnus Solutions/Cygwin/mounts v<version>" where <version> is the latest registry version associated with the Cygwin library (this version is not the same as the release number). The system-wide table is located under the same subkeys under HKEY_LOCAL_SYSTEM.

By default, the POSIX root `/` points to the system partition but it can be relocated to any directory in the Windows file system using the **mount** command. Whenever Cygwin generates a POSIX path from a Win32 one, it uses the longest matching prefix in the mount table. Thus, if `C:` is mounted as `/c` and also as `/`, then Cygwin would translate `C:/foo/bar` to `/c/foo/bar`.

Invoking **mount** without any arguments displays Cygwin's current set of mount points. In the following example, the C drive is the POSIX root and D drive is mapped to `/d`. Note that in this case, the root mount is a system-wide mount point that is visible to all users running Cygwin programs, whereas the `/d` mount is only visible to the current user.

Example 3-1. Displaying the current set of mount points

```
c:\cygnus\> mount
Device           Directory           Type        Flags
D:               /d                  user        textmode
C:               /                   system      textmode
```

You can also use the **mount** command to add new mount points, and the **umount** to delete them. See Section 3.6.7 and Section 3.6.9 for more information on how to use these utilities to set up your Cygwin POSIX file system.

Whenever Cygwin cannot use any of the existing mounts to convert from a particular Win32 path to a POSIX one, Cygwin will automatically default to an imaginary mount point under the default POSIX path `/cygdrive`. For example, if Cygwin accesses `Z:\foo` and the Z drive is not currently in the mount table, then `Z:\` would be automatically converted to `/cygdrive/Z`. The default prefix of `/cygdrive` may be changed (see the Section 3.6.7 for more information).

It is possible to assign some special attributes to each mount point. Automatically mounted partitions are displayed as "auto" mounts. Mounts can also be marked as either "textmode" or "binmode" – whether text files are read in the same manner as binary files by default or not (see Section 3.2 for more information on text and binary modes.

3.1.3. Cygwin Mount Table Strategies

Which set of mounts is right for a given Cygwin user depends largely on how closely you want to simulate a POSIX environment, whether you mix Windows and Cygwin programs, and how many drive letters you are using. If you want to be very POSIX-like (assuming "CygwinRoot" is the top directory of your Cygwin distribution), you may want to do something like this:

Example 3-2. POSIX-like mount setup

```
C:\> mount c:\Cygnus\CygwinRoot /
C:\> mount c:\ /c
C:\> mount d:\ /d
C:\> mount e:\ /cdrom
```

However, if you mix Windows and Cygwin programs a lot, you might want to create an "identity" mapping, so that conversions between the two (see Section 3.6.2) can be eliminated:

Example 3-3. Identity mount setup

```
C:\> mount c:\ \
```

```
C:\> mount d:\foo /foo
C:\> mount d:\bar /bar
C:\> mount e:\grill /grill
```

You'd have to repeat this for all top-level subdirectories on all drives, but then you'd always have the top-level directories available as the same names in both systems.

3.1.4. Additional Path-related Information

The **cygpath** program provides the ability to translate between Win32 and POSIX pathnames in shell scripts. See Section 3.6.2 for the details.

The HOME, PATH, and LD_LIBRARY_PATH environment variables are automatically converted from Win32 format to POSIX format (e.g. from `C:\cygnus\cygwin-b20\H-i586-cygwin32\bin` to `/bin`, if there was a mount from that Win32 path to that POSIX path) when a Cygwin process first starts.

Symbolic links can also be used to map Win32 pathnames to POSIX. For example, the command **ln -s //pollux/home/joe/data /data** would have about the same effect as creating a mount point from `//pollux/home/joe/data` to `/data` using **mount**, except that symbolic links cannot set the default file access mode. Other differences are that the mapping is distributed throughout the file system and proceeds by iteratively walking the directory tree instead of matching the longest prefix in a kernel table. Note that symbolic links will only work on network drives that are properly configured to support the "system" file attribute. Many do not do so by default (the Unix Samba server does not by default, for example).

3.2. Text and Binary modes

3.2.1. The Issue

On a UNIX system, when an application reads from a file it gets exactly what's in the file on disk and the converse is true for writing. The situation is different in the DOS/Windows world where a file can be opened in one of two modes, binary or text. In the binary mode the system behaves exactly as in UNIX. However in text mode there are major differences:

a. On writing in text mode, a NL (\n, ^J) is transformed into the sequence CR (\r, ^M) NL.
b. On reading in text mode, a CR followed by an NL is deleted and a ^Z character signals the end of file.

This can wreak havoc with the seek/fseek calls since the number of bytes actually in the file may differ from that seen by the application.

The mode can be specified explicitly as explained in the Programming section below. In an ideal DOS/Windows world, all programs using lines as records (such as **bash**, **make**, **sed** ...) would open files (and change the mode of their standard input and output) as text. All other programs (such as **cat**, **cmp**, **tr** ...) would use binary mode. In practice with Cygwin, programs that deal explicitly with object files specify binary mode (this is the case of **od**, which is helpful to diagnose CR problems). Most other programs (such as **cat**, **cmp**, **tr**) use the default mode.

3.2.2. The default Cygwin behavior

The Cygwin system gives us some flexibility in deciding how files are to be opened when the mode is not specified explicitly. The rules are evolving, this section gives the design goals.

a. If the file appears to reside on a file system that is mounted (i.e. if its pathname starts with a directory displayed by **mount**), then the default is specified by the mount flag. If the file is a symbolic link, the mode of the target file system applies.
b. If the file appears to reside on a file system that is not mounted (as can happen when the path contains a drive letter), the default is text.

c. Pipes and non-file devices are opened in binary mode, except if the CYGWIN environment variable contains `nobinmode`.

> **Warning!**
>
> In b20.1 of 12/98, a file will be opened in binary mode if any of the following conditions hold:
>
> 1. binary mode is specified in the open call
> 2. CYGWIN contains `binmode`
> 3. the file resides in a binary mounted partition
> 4. the file is not a disk file

d. When a Cygwin program is launched by a shell, its standard input, output and error are in binary mode if the CYGWIN variable contains `tty`, else in text mode, except if they are piped or redirected.

When redirecting, the Cygwin shells uses rules (a-c). For these shells the relevant value of CYGWIN is that at the time the shell was launched and not that at the time the program is executed. Non-Cygwin shells always pipe and redirect with binary mode. With non-Cygwin shells the commands **cat filename | program** and **program < filename** are not equivalent when `filename` is on a text-mounted partition.

3.2.3. Example

To illustrate the various rules, we provide scripts to delete CRs from files by using the **tr** program, which can only write to standard output. The script

```
#!/bin/sh
# Remove \r from the file given as argument
tr -d '\r' < "$1" > "$1".nocr
```

will not work on a text mounted systems because the \r will be reintroduced on writing. However scripts such as

```
#!/bin/sh
# Remove \r from the file given as argument
tr -d '\r' | gzip | gunzip > "$1".nocr
```

and the .bat file

```
REM Remove \r from the file given as argument
@echo off
tr -d \r < %1 > %1.nocr
```

work fine. In the first case (assuming the pipes are binary) we rely on **gunzip** to set its output to binary mode, possibly overriding the mode used by the shell. In the second case we rely on the DOS shell to redirect in binary mode.

3.2.4. Binary or text?

UNIX programs that have been written for maximum portability will know the difference between text and binary files and act appropriately under Cygwin. For those programs, the text mode default is a good choice. Programs included in official Cygnus distributions should work well in the default mode.

Text mode makes it much easier to mix files between Cygwin and Windows programs, since Windows programs will usually use the CRLF format. Unfortunately you may still have some problems with text mode. First, some of the utilities included with Cygwin do not yet specify binary mode when they should, e.g. **cat** will not work with binary files (input will stop at ^Z, CRs will be introduced in the output). Second, you will introduce CRs in text files you write, which can cause problems when moving them back to a UNIX system.

If you are mounting a remote file system from a UNIX machine, or moving files back and forth to a UNIX machine, you may want to access the files in binary mode. The text files found there will normally be in UNIX NL format, and you would want any files

put there by Cygwin programs to be stored in a format understood by UNIX. Be sure to remove CRs from all Makefiles and shell scripts and make sure that you only edit the files with DOS/Windows editors that can cope with and preserve NL terminated lines.

Note that you can decide this on a disk by disk basis (for example, mounting local disks in text mode and network disks in binary mode). You can also partition a disk, for example by mounting `c:` in text mode, and `c:\home` in binary mode.

3.2.5. Programming

In the `open()` function call, binary mode can be specified with the flag `O_BINARY` and text mode with `O_TEXT`. These symbols are defined in `fcntl.h`.

In the `fopen()` function call, binary mode can be specified by adding a `b` to the mode string. There is no direct way to specify text mode.

The mode of a file can be changed by the call `setmode(fd,mode)` where `fd` is a file descriptor (an integer) and `mode` is `O_BINARY` or `O_TEXT`. The function returns `O_BINARY` or `O_TEXT` depending on the mode before the call, and `EOF` on error.

3.3. File permissions

On Windows 9x systems, files are always readable, and Cygwin uses the native read-only mode to determine if they are writable. Files are considered to be executable if the filename ends with .bat, .com or .exe, or if its content starts with #!. Consequently **chmod** can only affect the "w" mode, it silently ignores actions involving the other modes. This means that **ls -l** needs to open and read files. It can thus be relatively slow.

Under NT, file permissions default to the same behavior as Windows 9x but there is optional functionality in Cygwin that can make file systems behave more like on UNIX systems. This is turned on by adding the "ntea" option to the CYGWIN environment variable.

When the "ntea" feature is activated, Cygwin will start with basic permissions as determined above, but can store POSIX file permissions in NT Extended Attributes. This feature works quite well on NTFS partitions because the attributes can be stored sensibly inside the normal NTFS filesystem structure. However, on a FAT partition, NT stores extended attributes in a flat file at the root of the partition called `EA DATA. SF`. This file can grow to extremely large sizes if you have a large number of files on the partition in question, slowing the system to a crawl. In addition, the `EA DATA. SF` file can only be deleted outside of Windows because of its "in use" status. For these reasons, the use of NT Extended Attributes is off by default in Cygwin. Finally, note that specifying "ntea" in CYGWIN has no effect under Windows 9x.

Under NT, the test "[-w filename]" is only true if filename is writable across the board, e.g. **chmod +w filename**.

3.4. Special filenames

3.4.1. DOS devices

Windows filenames invalid under Windows are also invalid under Cygwin. This means that base filenames such as `AUX`, `COM1`, `LPT1` or `PRN` (to name a few) cannot be used in a regular Cygwin Windows or POSIX path, even with an extension (`prn.txt`). However the special names can be used as filename extensions (`file.aux`). You can use the special names as you would under DOS, for example you can print on your default printer with the command **cat filename > PRN** (make sure to end with a Form Feed).

3.4.2. POSIX devices

There is no need to create a POSIX `/dev` directory as it is simulated within Cygwin automatically. It supports the following devices: `/dev/null`, `/dev/tty` and `/dev/comX` (the serial ports). These devices cannot be seen with the command **ls /dev**

although commands such as **ls /dev/tty** work fine.

* *FIXME: Are there other devices under /dev. What about the funny ones mounted by default, such as /dev/fd1. What do they really do?*

3.4.3. The .exe extension

Executable program filenames end with .exe but the .exe need not be included in the command, so that traditional UNIX names can be used. However, for programs that end in ".bat" and ".com", you cannot omit the extension.

As a side effect, the **ls filename** gives information about `filename.exe` if `filename.exe` exists and `filename` does not. In the same situation the function call `stat("filename",..)` gives information about `filename.exe`. The two files can be distinguished by examining their inodes, as demonstrated below.

```
C:\Cygnus\> ls *
a      a.exe     b.exe
C:\Cygnus\> ls -i a a.exe
445885548 a       435996602 a.exe
C:\Cygnus\> ls -i b b.exe
432961010 b       432961010 b.exe
```

If a shell script `myprog` and a program `myprog.exe` coexist in a directory, the program has precedence and is selected for execution of **myprog**.

The **gcc** compiler produces an executable named `filename.exe` when asked to produce `filename`. This allows many makefiles written for UNIX systems to work well under Cygwin.

Unfortunately, the **install** and **strip** commands do distinguish between `filename` and `filename.exe`. They fail when working on a non-existing `filename` even if `filename.exe` exists, thus breaking some makefiles. This problem can be solved by writing **install** and **strip** shell scripts to provide the extension ".exe" when needed.

3.4.4. The @pathnames

To circumvent the limitations on shell line length in the native Windows command shells, Cygwin programs expand their arguments starting with "@" in a special way. If a file `pathname` exists, the argument `@pathname` expands recursively to the content of `pathname`. Double quotes can be used inside the file to delimit strings containing blank space. Embedded double quotes must be repeated. In the following example compare the behaviors of the bash built-in **echo** and of the program **/bin/echo**.

Example 3-4. Using @pathname

```
/Cygnus$ echo 'This is "a long" line' > mylist
/Cygnus$ echo @mylist
@mylist
/Cygnus$ /bin/echo @mylist
This is a long line
/Cygnus$ rm mylist
/Cygnus$ /bin/echo @mylist
@mylist
```

3.5. The CYGWIN environment variable

The CYGWIN environment variable is used to configure many global settings for the Cygwin runtime system. It contains the options listed below, separated by blank characters. Many options can be turned off by prefixing with `no`.

- *(no)binmode* - if set, non-disk (e.g. pipe and COM ports) file opens default to binary mode (no CR/LF/Ctrl-Z translations) instead of text mode. Defaults to set (binary mode). This option must be set before starting a Cygwin shell to have an effect on redirection.

> **Warning!**
>
> If set in 12/98 b20.1, all files always open in binary mode.

- *(no)envcache* - If set, environment variable conversions (between Win32 and POSIX) are cached. Note that this is may cause problems if the mount table changes, as the cache is not invalidated and may contain values that depend on the previous mount table contents. Defaults to set.
- *(no)export* - if set, the final values of these settings are re-exported to the environment as $CYGWIN again.
- *(no)glob* - if set, command line arguments containing UNIX-style file wildcard characters (brackets, question mark, asterisk, escaped with \) are expanded into lists of files that match those wildcards. This is applicable only to programs running from a DOS command line prompt. Default is set.
- *(no)ntea* - if set, use the full NT Extended Attributes to store UNIX-like inode information. This option only operates under Windows NT. Defaults to not set.

> **Warning!**
>
> This may create additional *large* files on non-NTFS partitions.

- *(no)ntsec* - if set, use the NT security model to set UNIX-like permissions on files and processes. The file permissions can only be set on NTFS partitions. FAT and SAMBA doesn't support the NT file security. For more information, read the documentation in [ntsec.sgml].
- *(no)reset_com* - if set, serial ports are reset to 9600-8-N-1 with no flow control when used. This is done at open time and when handles are inherited. Defaults to set.
- *strace=n[:cache][,filename]* - configures system tracing. Off by default, setting various bits in `n` (a bit flag) enables various types of system messages. Setting `n` to 1 enables most messages. Other values can be found in `sys/strace.h`. The `:cache`

option lets you specify how many lines to cache before flushing the output (example: `strace=1:20`). The `filename` option lets you send the messages to a file instead of the screen.

- *(no)strip_title* - if set, strips the directory part off the window title, if any. Default is not set.
- *(no)title* - if set, the title bar reflects the name of the program currently running. Default is not set. Note that under Win9x the title bar is always enabled and it is stripped by default, but this is because of the way Win9x works. In order not to strip, specify `title` or `title nostrip_title`.
- *(no)tty* - if set, Cygwin enables extra support (i.e., termios) for UNIX-like ttys. It is not compatible with some Windows programs. Defaults to not set, in which case the tty is opened in text mode with ^Z as EOF. Note that this has been changed such that ^D works as expected instead of ^Z, and is settable via stty. This option must be specified before starting a Cygwin shell and it cannot be changed in the shell.

3.6. Cygwin Utilities

Cygwin comes with a number of command-line utilities that are used to manage the UNIX emulation portion of the Cygwin environment. While many of these reflect their UNIX counterparts, each was written specifically for Cygwin.

3.6.1. cygcheck

```
Usage: cygcheck [-s] [-v] [-r] [-h] [program ...]
  -s = system information
  -v = verbose output (indented) (for -s or programs)
  -r = registry search (requires -s)
  -h = give help about the info
You must at least give either -s or a program name
```

The **cygcheck** program is a diagnostic utility that examines your system and reports the information that is significant to the proper operation of Cygwin programs. It can give information about a specific program (or program) you are trying to run, general system information, or both. If you list one or more programs on the command line, it will diagnose the runtime environment of that program or programs. If you specify the `-s` option, it will give general system information. If you specify `-s` and list one or more programs on the command line, it reports on both.

The **cygcheck** program should be used to send information about your system to Cygnus for troubleshooting (if your support representative requests it). When asked to run this command, include all the options plus any commands you are having trouble with, and save the output so that you can mail it to Cygnus, like this:

```
C:\Cygnus> cygcheck -s -v -r -h > tocygnus.txt
```

The `-v` option causes the output to be more verbose. What this means is that additional information will be reported which is usually not interesting, such as the internal version numbers of DLLs, additional information about recursive DLL usage, and if a file in one directory in the PATH also occurs in other directories on the PATH.

The `-r` option causes **cygcheck** to search your registry for information that is relevent to Cygnus programs. These registry entries are the ones that have "Cygnus" in the name. If you are paranoid about privacy, you may remove information from this report, but please keep in mind that doing so makes it harder for Cygnus to diagnose your problems.

The `-h` option prints additional helpful messages in the report, at the beginning of each section. It also adds table column headings. While this is useful information, it also adds some to the size of the report, so if you want a compact report or if you know what everything is already, just leave this out.

3.6.2. cygpath

```
Usage: cygpath [-p|-path] (-u|-unix)|(-w|-windows) filename
       cygpath [-v|-version]
```

```
     cygpath [-W|-windir|-S|-sysdir]
  -u|-unix     print UNIX form of filename
  -w|-windows  print Windows form of filename
  -p|-path     filename argument is a path
  -v|-version  print program version
  -W|-windir   print windows directory
  -S|-sysdir   print system directory
```

The **cygpath** program is a utility that converts Windows native filenames to Cygwin POSIX-style pathnames and back. It can be used when a Cygwin program needs to pass a file name to a native Windows program, or expects to get a file name from a native Windows program. You may use the long or short option names interchangeably, even though only the short ones are described here.

The `-u` and `-w` options indicate whether you want a conversion from Windows to UNIX (POSIX) format (`-u`) or a conversion from UNIX (POSIX) to Windows format (`-w`). You must give exactly one of these. To give neither or both is an error.

The `-p` option means that you want to convert a path-style string rather than a single filename. For example, the PATH environment variable is semicolon-delimited in Windows, but colon-delimited in UNIX. By giving `-p` you are instructing **cygpath** to convert between these formats.

Example 3-5. Example cygpath usage

```
#!/bin/sh
for i in ‘echo *.exe | sed ’s/\.exe/cc/’‘
do
  notepad ‘cygpath -w $i‘
done
```

3.6.3. kill

```
Usage: kill [-sigN] pid1 [pid2 ...]
```

The **kill** program allows you to send arbitrary signals to other Cygwin programs. The usual purpose is to end a running program from some other window when ^C won't work, but you can also send program-specified signals such as SIGUSR1 to trigger actions within the program, like enabling debugging or re-opening log files. Each program defines the signals they understand.

Note that the "pid" values are the Cygwin pids, not the Windows pids. To get a list of running programs and their Cygwin pids, use the Cygwin **ps** program.

To send a specific signal, use the `-signN` option, either with a signal number or a signal name (minus the "SIG" part), like these examples:

Example 3-6. Specifying signals with the kill command

```
$ kill 123
$ kill -1 123
$ kill -HUP 123
```

Here is a list of available signals, their numbers, and some commentary on them, from the file `<sys/signal.h>`, which should be considered the official source of this information.

```
SIGHUP       1    hangup
SIGINT       2    interrupt
SIGQUIT      3    quit
SIGILL       4    illegal instruction (not reset when caught)
SIGTRAP      5    trace trap (not reset when caught)
SIGABRT      6    used by abort
SIGEMT       7    EMT instruction
SIGFPE       8    floating point exception
SIGKILL      9    kill (cannot be caught or ignored)
SIGBUS      10    bus error
SIGSEGV     11    segmentation violation
SIGSYS      12    bad argument to system call
SIGPIPE     13    write on a pipe with no one to read it
SIGALRM     14    alarm clock
SIGTERM     15    software termination signal from kill
```

```
SIGURG      16    urgent condition on IO channel
SIGSTOP     17    sendable stop signal not from tty
SIGTSTP     18    stop signal from tty
SIGCONT     19    continue a stopped process
SIGCHLD     20    to parent on child stop or exit
SIGCLD      20    System V name for SIGCHLD
SIGTTIN     21    to readers pgrp upon background tty read
SIGTTOU     22    like TTIN for output if (tp->t_local&LTOSTOP)
SIGIO       23    input/output possible signal
SIGPOLL     23    System V name for SIGIO
SIGXCPU     24    exceeded CPU time limit
SIGXFSZ     25    exceeded file size limit
SIGVTALRM   26    virtual time alarm
SIGPROF     27    profiling time alarm
SIGWINCH    28    window changed
SIGLOST     29    resource lost (eg, record-lock lost)
SIGUSR1     30    user defined signal 1
SIGUSR2     31    user defined signal 2
```

3.6.4. mkgroup

```
usage: mkgroup <options> [domain]
This program prints group information to stdout
Options:\n");
    -l,-
local           print pseudo group information if there is
                        no domain
    -d,-
domain          print global group information from the domain
                        specified (or from the current do-
main if there is
                        no domain specified)
    -?,-help           print this message
```

The **mkgroup** program can be used to help configure your Windows system to be more UNIX-like by creating an initial `/etc/group` substitute (some commands need this file) from your system information. It only works on NT. To initially set up your machine, you'd do something like this:

Example 3-7. Setting up the groups file

```
$ mkdir /etc
$ mkgroup -l > /etc/group
```

Note that this information is static. If you change the group information in your system, you'll need to regenerate the group file for it to have the new information.

The `-d` and `-l` options allow you to specify where the information comes from, either the local machine or the default (or given) domain.

3.6.5. mkpasswd

```
Usage: mkpasswd [options] [domain]
This program prints a /etc/passwd file to stdout
Options are
   -l,-local              print local accounts
   -d,-
domain             print domain accounts (from current domain
                         if no domain specified
   -g,-local-groups       print local group information too
   -?,-help               displays this message
This program does only work on Windows NT
```

The **mkpasswd** program can be used to help configure your Windows system to be more UNIX-like by creating an initial `/etc/passwd` substitute (some commands need this file) from your system information. It only works on NT. To initially set up your machine, you'd do something like this:

Example 3-8. Setting up the passwd file

```
$ mkdir /etc
$ mkpasswd -l > /etc/passwd
```

Note that this information is static. If you change the user information in your system, you'll need to regenerate the passwd file for it to have the new information.

The `-d` and `-l` options allow you to specify where the information comes from, either the local machine or the default (or given) domain.

3.6.6. passwd

```
Usage passwd [name]
      passwd [-x max] [-n min] [-i inact] [-L len]
      passwd {-l|-u|-S} name
  -x max   set max age of passwords
  -n min   set min age of passwords
  -i inact disables account after inact days of expiry
  -L len   set min password length
  -l       lock an account
  -u       unlock an account
  -S       show account information
```

passwd changes passwords for user accounts. A normal user may only change the password for their own account, the administrators may change the password for any account. **passwd** also changes account information, such as password expiry dates and intervals.

Password changes: The user is first prompted for their old password, if one is present. This password is then encrypted and compared against the stored password. The user has only one chance to enter the correct password. The administrators are permitted to bypass this step so that forgotten passwords may be changed.

The user is then prompted for a replacement password. **passwd** will prompt again and compare the second entry against the first. Both entries are require to match in order for the password to be changed.

After the password has been entered, password aging information is checked to see if the user is permitted to change their password at this time. If not, **passwd** refuses to change the password and exits.

Password expiry and length: The password aging information may be changed by the administrators with the `-x`, `-n` and `-i` options. The `-x` option is used to set the maximum number of days a password remains valid. After *max* days, the password is required to be changed. The `-n` option is used to set the minimum number of days before a password may be changed. The user will not be permitted to change the password until *min* days have elapsed. The `-i` option is used to disable an account after the password has been expired for a number of days. After a user account has had an expired password for *inact* days, the user may no longer sign on to the account. Allowed values for the above options are 0 to 999. The `-L` option sets the minimum length of allowed passwords for users, which doesn't belong to the administrators group, to *len* characters. Allowed values for the minimum password length are 0 to 14. In any of the above cases, a value of 0 means 'no restrictions'.

Account maintenance: User accounts may be locked and unlocked with the `-l` and `-u` flags. The `-l` option disables an account. The `-u` option re-enables an account.

The account status may be given with the `-S` option. The status information is self explanatory.

Limitations: Users may not be able to change their password on some systems.

3.6.7. mount

```
Usage mount
      mount [-bfs] <win32path> <posixpath>
      mount [-bs] -change-cygdrive-prefix<posixpath>
      mount -import-old-mounts
```

```
  -b = text files are equivalent to binary files (newline = \n)
  -x = files in the mounted directory are automati-
cally given execute permission.
  -f = force mount, don't warn about miss-
ing mount point directories
  -s = add mount point to system-wide registry location
  -change-automount-
prefix = change path prefix used for automatic mount points
  -import-old-mounts = copy old registry mount ta-
ble mounts into the current mount areas

  When invoked without any arguments, mount displays the cur-
rent mount table.
```

The **mount** program is used to map your drives and shares onto Cygwin's simulated POSIX directory tree, much like as is done by mount commands on typical UNIX systems. Please see Section 3.1.2 for more information on the concepts behind the Cygwin POSIX file system and strategies for using mounts.

3.6.7.1. Using mount

If you just type **mount** with no parameters, it will display the current mount table for you.

Example 3-9. Displaying the current set of mount points

```
c:\cygnus\> mount
Device           Directory           Type        Flags
D:               /d                  user        textmode
C:               /                   system      textmode
```

In this example, the C drive is the POSIX root and D drive is mapped to `/d`. Note that in this case, the root mount is a system-wide mount point that is visible to all users running Cygwin programs, whereas the `/d` mount is only visible to the current user.

The **mount** utility is also the mechanism for adding new mounts to the mount table. The following example demonstrates how to mount the directory `C:\cygnus\cygwin-b20\H-i586-cygwin32\bin` to `/bin` and the network directory `\\pollux\home\joe\data` to `/data`. `/bin` is assumed to already exist.

Example 3-10. Adding mount points

```
c:\cygnus\> ls /bin /data
ls: /data: No such file or directory
c:\cygnus\> mount C:\cygnus\cygwin-b20\H-i586-cygwin32\bin /bin
c:\cygnus\> mount \\pollux\home\joe\data /data
Warning: /data does not exist!
c:\cygnus\> mount
Device           Directory           Type        Flags
\\pollux\home\joe\data   /data       user        textmode
C:\cygnus\cygwin-b20\H-i586-
cygwin32\bin   /bin   user   textmode
D:               /d                  user        textmode
\\.\tape1:       /dev/st1            user        textmode
\\.\tape0:       /dev/st0            user        textmode
\\.\b:           /dev/fd1            user        textmode
\\.\a:           /dev/fd0            user        textmode
C:               /                   system      textmode
c:\cygnus\> ls /bin/sh
/bin/sh
```

Note that **mount** was invoked from the Windows command shell in the previous example. In many Unix shells, including bash, it is legal and convenient to use the forward "/" in Win32 pathnames since the "\" is the shell's escape character.

The "-s" flag to **mount** is used to add a mount in the system-wide mount table used by all Cygwin users on the system, instead of the user-specific one. System-wide mounts are displayed by **mount** as being of the "system" type, as is the case for the `/` partition in the last example. Under Windows NT, only those users with Administrator priviledges are permitted to modify the system-wide mount table.

Note that a given POSIX path may only exist once in the user table and once in the global, system-wide table. Attempts to replace the mount will fail with a busy error. The "-f" (force) flag causes the old mount to be silently replaced with the new one. It will also silence warnings about the non-existence of directories at the Win32 path location.

The "-b" flag is used to instruct Cygwin to treat binary and text files in the same manner by default. Binary mode mounts are marked as "binmode" in the Flags column of **mount** output. By default, mounts are in text mode ("textmode" in the Flags column).

The "-x" flag is used to instruct Cygwin that the mounted file is "executable". If the "-x" flag is used with a directory then all files in the directory are executable. Files ending in certain extensions (.exe, .com, .bat, .cmd) are assumed to be executable by default. Files whose first two characters begin with '#!' are also considered to be executable. This option allows other files to be marked as executable and avoids the overhead of opening each file to check for a '#!'.

3.6.7.2. Cygdrive mount points

Whenever Cygwin cannot use any of the existing mounts to convert from a particular Win32 path to a POSIX one, Cygwin will, instead, convert to a POSIX path using a default mount point: `/cygdrive`. For example, if Cygwin accesses `Z:\foo` and the Z drive is not currently in the mount table, then `Z:\` will be accessible as `/cygdrive/Z`. The default prefix of `/cygdrive` may be changed via the Section 3.6.7 command.

The **mount** utility can be used to change this default automount prefix through the use of the "–change-cygdrive-prefix" flag. In the following example, we will set the automount prefix to `/`:

Example 3-11. Changing the default prefix

```
c:\cygnus\> mount -change-cygdrive-prefix /
```

Note that you if you set a new prefix in this manner, you can specify the "-s" flag to make this the system-wide default prefix. By default, the cygdrive-prefix applies only to the current user. In the same way, you can specify the "-b" flag such that all new

automounted filesystems default to binary mode file accesses.

3.6.7.3. Limitations

Limitations: there is a hard-coded limit of 30 mount points. Also, although you can mount to pathnames that do not start with "/", there is no way to make use of such mount points.

Normally the POSIX mount point in Cygwin is an existing empty directory, as in standard UNIX. If this is the case, or if there is a place-holder for the mount point (such as a file, a symbolic link pointing anywhere, or a non-empty directory), you will get the expected behavior. Files present in a mount point directory before the mount become invisible to Cygwin programs.

It is sometimes desirable to mount to a non-existent directory, for example to avoid cluttering the root directory with names such as `a`, `b`, `c` pointing to disks. Although **mount** will give you a warning, most everything will work properly when you refer to the mount point explicitly. Some strange effects can occur however. For example if your current working directory is `/dir`, say, and `/dir/mtpt` is a mount point, then `mtpt` will not show up in an **ls** or **echo *** command and **find .** will not find `mtpt`.

3.6.8. ps

```
Usage ps [-aefl] [-u uid]
  -f       show process uids, ppids
  -l       show process uids, ppids, pgids, winpids
  -u uid   list processes owned by uid
  -a, -e   show processes of all users
```

The **ps** program gives the status of all the Cygwin processes running on the system (ps = "process status"). Due to the limitations of simulating a POSIX environment under Windows, there is little information to give. The PID column is the process ID you

need to give to the **kill** command. The WINPID column is the process ID that's displayed by NT's Task Manager program.

3.6.9. umount

```
Usage umount [-s] <posixpath>
-s = remove mount point from system-wide registry location

-remove-all-mounts = remove all mounts
-remove-auto-mounts = remove all automatically mounted mounts
-remove-user-mounts = remove all mounts in the cur-
rent user mount registry area, including auto mounts
-remove-system-mounts = Remove all mounts in the system-
wide mount registry area
```

The **umount** program removes mounts from the mount table. If you specify a POSIX path that corresponds to a current mount point, **umount** will remove it from the user-specific registry area. The -s flag may be used to specify removing the mount from the system-wide registry area instead (Administrator priviledges are required).

The **umount** utility may also be used to remove all mounts of a particular type. With the extended options it is possible to remove all mounts, all automatically-mounted mounts, all mounts in the current user's registry area, or all mounts in the system-wide registry area (with Administrator priviledges).

See Section 3.6.7) for more information on the mount table.

3.6.10. strace

```
Usage strace [-m mask] [-o output-file] [ft] program [args...]

-m mask   mask for reporting cygwin events (default 1)
-o output-file
output file to hold strace events (default stderr)
```

```
-f   follow forked subprocesses
-t   convert Win32 error messages to text
-s   remove mount point from system-wide registry location
```

The **strace** program executes a program, and optionally the children of the program, reporting any Cygwin DLL output from the program(s) to file. This program is mainly useful for debugging the Cygwin DLL itself. The mask argument is a hexadecimal string signifying which events should be reported. The valid bits to set are as follows:

```
  Bit Explanation
0x00000001 All strace output is collected
0x00000002 Unusual or weird phenomenon
0x00000010 System calls
0x00000020 argv/envp printout at startup
0x00000040 Information useful for DLL debugging
0x00000080 Paranoid information
0x00000100 Termios debbugging
0x00000200 Select() function debugging
0x00000400 Window message debugging
0x00000800 Signal and process handling
0x00001000 Very minimal strace output
0x00020000 Malloc calls
0x00040000 Thread locking calls
```

3.6.11. regtool

```
regtool -h  - print this message
regtool [-v] list [key]  - list subkeys and values
regtool [-v] add [key\subkey]  - add new subkey
regtool [-v] remove [key]  - remove key
regtool [-v|-q] check [key]  - exit 0 if key exists, 1 if not
regtool [-i|-s|-e|-m] set [key\value] [data ...]  - set value
     -i=integer -s=string -e=expand-string -m=multi-string
regtool [-v] unset [key\value]  - removes value from key
regtool [-q] get [key\value]  - prints value to stdout
```

```
     -q=quiet, no error msg, just re-
turn nonzero exit if key/value missing
keys are like \prefix\key\key\key\value, where prefix is any of:
   root     HKCR  HKEY_CLASSES_ROOT
   config   HKCC  HKEY_CURRENT_CONFIG
   user     HKCU  HKEY_CURRENT_USER
   machine  HKLM  HKEY_LOCAL_MACHINE
   users    HKU   HKEY_USERS
example: \user\software\Microsoft\Clock\iFormat
```

The **regtool** program allows shell scripts to access and modify the Windows registry. Note that modifying the Windows registry is dangerous, and carelessness here can result in an unusable system. Be careful.

The `-v` option means "verbose". For most commands, this causes additional or lengthier messages to be printed. Conversely, the `-q` option supresses error messages, so you can use the exit status of the program to detect if a key exists or not (for example).

The `list` command lists the subkeys and values belonging to the given key. The `add` command adds a new key. The `remove` command removes a key. Note that you may need to remove everything in the key before you may remove it, but don't rely on this stopping you from accidentally removing too much. The `check` command checks to see if a key exists (the exit code of the program is zero if it does, nonzero if it does not).

The `set` command sets a value within a key. `-i` means the value is an integer (DWORD). `-s` means the value is a string. `-e` means it's an expanding string (it contains embedded environment variables). `-m` means it's a multi-string (list). If you don't specify one of these, it tries to guess the type based on the value you give. If it looks like a number, it's a number. If it starts with a percent, it's an expanding string. If you give multiple values, it's a multi-string. Else, it's a regular string.

The `unset` command removes a value from a key. The `get` command gets the value of a value of a key, and prints it (and nothing else) to stdout. Note: if the value doesn't exist, an error message is printed and the program returns a non-zero exit code. If you give `-q`, it doesn't print the message but does return the non-zero exit code.

Chapter 4. Programming with Cygwin

4.1. Using GCC with Cygwin

4.1.1. Console Mode Applications

Use gcc to compile, just like under UNIX. Refer to the GCC User's Guide for information on standard usage and options. Here's a simple example:

Example 4-1. Building Hello World with GCC

```
C:\cygnus\> gcc hello.c -o hello.exe
C:\cygnus\> hello.exe
Hello, World

C:\cygnus\>
```

4.1.2. GUI Mode Applications

Cygwin allows you to build programs with full access to the standard Windows 32-bit API, including the GUI functions as defined in any Microsoft or off-the-shelf publication. However, the process of building those applications is slightly different, as you'll be using the GNU tools instead of the Microsoft tools.

For the most part, your sources won't need to change at all. However, you should remove all __export attributes from functions and replace them like this:

```
int foo (int) __attribute__ ((__dllexport__));

int
foo (int i)
```

For most cases, you can just remove the __export and leave it at that. For convenience sake, you might want to include the following code snippet when compiling GUI programs. If you don’t, you will want to add "-e _mainCRTStartup" to your link line in your Makefile.

```
#ifdef __CYGWIN__
WinMainCRTStartup() { mainCRTStartup(); }
#endif
```

The Makefile is similar to any other UNIX-like Makefile, and like any other Cygwin makefile. The only difference is that you use **gcc -mwindows** to link your program into a GUI application instead of a command-line application. Here’s an example:

```
myapp.exe : myapp.o myapp.res
gcc -mwindows myapp.o myapp.res -o $@

myapp.res : myapp.rc resource.h
windres $< -O coff -o $@
```

Note the use of `windres` to compile the Windows resources into a COFF-format `.res` file. That will include all the bitmaps, icons, and other resources you need, into one handy object file. Normally, if you omitted the "-O coff" it would create a Windows `.res` format file, but we can only link COFF objects. So, we tell `windres` to produce a COFF object, but for compatibility with the many examples that assume your linker can handle Windows resource files directly, we maintain the `.res` naming convention. For more information on `windres`, consult the Binutils manual.

4.2. Debugging Cygwin Programs

When your program doesn’t work right, it usually has a "bug" in it, meaning there’s something wrong with the program itself that is causing unexpected results or crashes. Diagnosing these bugs and fixing them is made easy by special tools called *debuggers*. In the case of Cygwin, the debugger is GDB, which stands for "GNU DeBugger". This

tool lets you run your program in a controlled environment where you can investigate the state of your program while it is running or after it crashes. Crashing programs sometimes create "core" files. In Cygwin these are regular text files that cannot be used directly by GDB.

Before you can debug your program, you need to prepare your program for debugging. What you need to do is add `-g` to all the other flags you use when compiling your sources to objects.

Example 4-2. Compiling with -g

```
$ gcc -g -O2 -c myapp.c
$ gcc -g myapp.c -o myapp
```

What this does is add extra information to the objects (they get much bigger too) that tell the debugger about line numbers, variable names, and other useful things. These extra symbols and debugging information give your program enough information about the original sources so that the debugger can make debugging much easier for you.

In Windows versions of GNUPro, GDB comes with a full-featured graphical interface. In Cygwin Net distributions, GDB is only available as a command-line tool. To invoke GDB, simply type **gdb myapp.exe** at the command prompt. It will display some text telling you about itself, then `(gdb)` will appear to prompt you to enter commands. Whenever you see this prompt, it means that gdb is waiting for you to type in a command, like **run** or **help**. Oh `:-)` type **help** to get help on the commands you can type in, or read the [GDB User's Manual] for a complete description of GDB and how to use it.

If your program crashes and you're trying to figure out why it crashed, the best thing to do is type **run** and let your program run. After it crashes, you can type **where** to find out where it crashed, or **info locals** to see the values of all the local variables. There's also a **print** that lets you look at individual variables or what pointers point to.

If your program is doing something unexpected, you can use the **break** command to tell gdb to stop your program when it gets to a specific function or line number:

Example 4-3. "break" in gdb

```
(gdb) break my_function
(gdb) break 47
```

Now, when you type **run** your program will stop at that "breakpoint" and you can use the other gdb commands to look at the state of your program at that point, modify variables, and **step** through your program's statements one at a time.

Note that you may specify additional arguments to the **run** command to provide command-line arguments to your program. These two cases are the same as far as your program is concerned:

Example 4-4. Debugging with command line arguments

```
$ myprog -t foo -queue 47

$ gdb myprog
(gdb) run -t foo -queue 47
```

4.3. Building and Using DLLs

DLLs are Dynamic Link Libraries, which means that they're linked into your program at run time instead of build time. There are three parts to a DLL:

- the exports
- the code and data
- the import library

The code and data are the parts you write - functions, variables, etc. All these are merged together, like if you were building one big object files, and put into the dll. They are not put into your .exe at all.

The exports contains a list of functions and variables that the dll makes available to other programs. Think of this as the list of "global" symbols, the rest being hidden.

Normally, you'd create this list by hand with a text editor, but it's possible to do it automatically from the list of functions in your code. The `dlltool` program creates the exports section of the dll from your text file of exported symbols.

The import library is a regular UNIX-like `.a` library, but it only contains the tiny bit of information needed to tell the OS how your program interacts with ("imports") the dll. This information is linked into your `.exe`. This is also generated by `dlltool`.

4.3.1. Building DLLs

OK, let's go through a simple example of how to build a dll. For this example, we'll use a single file `myprog.c` for the program (`myprog.exe`) and a single file `mydll.c` for the contents of the dll (`mydll.dll`).

Now compile everything to objects:

```
gcc -c myprog.c
gcc -c mydll.c
```

Unfortunately, the process for building a dll is, well, convoluted. You have to run five commands, like this:

```
gcc -s -Wl,-base-file,mydll.base -o mydll.dll mydll.o -Wl,-
e,_mydll_init@12
dlltool -base-file mydll.base -def mydll.def -output-
exp mydll.exp -dllname mydll.dll
gcc -s -Wl,-base-file,mydll.base,mydll.exp -
o mydll.dll mydll.o -Wl,-e,_mydll_init@12
dlltool -base-file mydll.base -def mydll.def -output-
exp mydll.exp -dllname mydll.dll
gcc -Wl,mydll.exp -o mydll.dll mydll.o -Wl,-e,_mydll_init@12
```

The extra steps give `dlltool` the opportunity to generate the extra sections (exports and relocation) that a dll needs. After this, you build the import library:

```
dlltool -def mydll.def -dllname mydll.dll -output-lib mydll.a
```

Now, when you build your program, you link against the import library:

```
gcc -o myprog myprog.o mydll.a
```

Note that we linked with **-e _mydll_init@12**. This tells the OS what the DLL's "entry point" is, and this is a special function that coordinates bringing the dll to life withing the OS. The minimum function looks like this:

```
#include <windows.h>

int WINAPI
mydll_init(HANDLE h, DWORD reason, void *foo)
{
  return 1;
}
```

4.3.2. Linking Against DLLs

If you have an existing DLL already, you need to build a Cygwin-compatible import library (The supplied ones should work, but you might not have them) to link against. Unfortunately, there is not yet any tool to do this automatically. However, you can get most of the way by creating a .def file with these commands (you might need to do this in `bash` for the quoting to work correctly):

```
echo EXPORTS > foo.def
nm foo.dll | grep ' T _' | sed 's/.* T _//' » foo.def
```

Note that this will only work if the DLL is not stripped. Otherwise you will get an error message: "No symbols in foo.dll".

Once you have the `.def` file, you can create an import library from it like this:

```
dlltool -def foo.def -dllname foo.dll -output-lib foo.a
```

4.4. Defining Windows Resources

`windres` reads a Windows resource file (`*.rc`) and converts it to a res or coff file. The syntax and semantics of the input file are the same as for any other resource compiler, so please refer to any publication describing the Windows resource format for details. Also, the `windres` program itself is fully documented in the Binutils manual. Here's an example of using it in a project:

```
myapp.exe : myapp.o myapp.res
gcc -mwindows myapp.o myapp.res -o $@

myapp.res : myapp.rc resource.h
windres $< -O coff -o $@
```

What follows is a quick-reference to the syntax `windres` supports.

```
id ACCELERATORS suboptions
BEG
"^C" 12
"Q" 12
65 12
65 12 , VIRTKEY ASCII NOINVERT SHIFT CONTROL ALT
65 12 , VIRTKEY, ASCII, NOINVERT, SHIFT, CONTROL, ALT
(12 is an acc_id)
END

SHIFT, CONTROL, ALT require VIRTKEY


id BITMAP memflags "filename"
memflags defaults to MOVEABLE


id CURSOR memflags "filename"
memflags defaults to MOVEABLE,DISCARDABLE
```

```
id DIALOG memflags exstyle x,y,width,height styles BEG con-
trols END
id DIALOGEX memflags exstyle x,y,width,height styles BEG con-
trols END
id DIALOGEX mem-
flags exstyle x,y,width,height,helpid styles BEG controls END

memflags defaults to MOVEABLE
exstyle may be EXSTYLE=number
styles: CAPTION "string"
CLASS id
STYLE  FOO | NOT FOO | (12)
EXSTYLE number
FONT number, "name"
FONT number, "name",weight,italic
MENU id
CHARACTERISTICS number
LANGUAGE number,number
VERSIONK number
controls:
AUTO3STATE params
AUTOCHECKBOX params
AUTORADIOBUTTON params
BEDIT params
CHECKBOX params
COMBOBOX params
CONTROL ["name",] id, class, style, x,y,w,h [,exstyle] [data]
CONTROL ["name",] id, class, style, x,y,w,h, exstyle, helpid [data]
CTEXT params
DEFPUSHBUTTON params
EDITTEXT params
GROUPBOX params
HEDIT params
ICON ["name",] id, x,y [data]
ICON ["name",] id, x,y,w,h, style, exstyle [data]
ICON ["name",] id, x,y,w,h, style, exstyle, helpid [data]
IEDIT params
```

```
LISTBOX params
LTEXT params
PUSHBOX params
PUSHBUTTON params
RADIOBUTTON params
RTEXT params
SCROLLBAR params
STATE3 params
USERBUTTON "string", id, x,y,w,h, style, exstyle
params:
["name",] id, x, y, w, h, [data]
["name",] id, x, y, w, h, style [,exstyle] [data]
["name",] id, x, y, w, h, style, exstyle, helpid [data]

[data] is op-
tional BEG (string|number) [,(string|number)] (etc) END


id FONT memflags "filename"
memflags defaults to MOVEABLE|DISCARDABLE

id ICON memflags "filename"
memflags defaults to MOVEABLE|DISCARDABLE

LANGUAGE num,num

id MENU options BEG items END
items:
"string", id, flags
SEPARATOR
POPUP "string" flags BEG menuitems END
flags:
CHECKED
GRAYED
HELP
INACTIVE
MENUBARBREAK
```

```
MENUBREAK

id MENUEX suboptions BEG items END
items:
MENUITEM "string"
MENUITEM "string", id
MENUITEM "string", id, type [,state]
POPUP "string" BEG items END
POPUP "string", id BEG items END
POPUP "string", id, type BEG items END
POPUP "string", id, type, state [,helpid] BEG items END

id MESSAGETABLE memflags "filename"
memflags defaults to MOVEABLE

id RCDATA subop-
tions BEG (string|number) [,(string|number)] (etc) END

STRINGTABLE suboptions BEG strings END
strings:
id "string"
id, "string"

(User data)
id id subop-
tions BEG (string|number) [,(string|number)] (etc) END

id VERSIONINFO stuffs BEG verblocks END
stuffs: FILEVERSION num,num,num,num
PRODUCTVERSION num,num,num,num
FILEFLAGSMASK num
FILEOS num
FILETYPE num
FILESUBTYPE num
verblocks:
BLOCK "StringFileInfo" BEG BLOCK BEG vervals END END
BLOCK "VarFileInfo" BEG BLOCK BEG vertrans END END
```

```
vervals: VALUE "foo","bar"
vertrans: VALUE num,num



suboptions:
memflags
CHARACTERISTICS num
LANGUAGE num,num
VERSIONK num

memflags are MOVE-
ABLE/FIXED PURE/IMPURE PRELOAD/LOADONCALL DISCARDABLE
```