

# **Cygwin User's Guide**

---

Copyright © 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012 Red Hat, Inc.

Permission is granted to make and distribute verbatim copies of this documentation provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

---

# Contents

<b>1</b>	<b>Cygwin Overview</b>	<b>1</b>
1.1	What is it? . . . . .	1
1.2	Quick Start Guide for those more experienced with Windows . . . . .	1
1.3	Quick Start Guide for those more experienced with UNIX . . . . .	1
1.4	Are the Cygwin tools free software? . . . . .	2
1.5	A brief history of the Cygwin project . . . . .	2
1.6	Highlights of Cygwin Functionality . . . . .	3
1.6.1	Introduction . . . . .	3
1.6.2	Permissions and Security . . . . .	3
1.6.3	File Access . . . . .	3
1.6.4	Text Mode vs. Binary Mode . . . . .	4
1.6.5	ANSI C Library . . . . .	5
1.6.6	Process Creation . . . . .	5
1.6.6.1	Problems with process creation . . . . .	5
1.6.7	Signals . . . . .	6
1.6.8	Sockets . . . . .	6
1.6.9	Select . . . . .	7
1.7	What's new and what changed in Cygwin 1.7 . . . . .	7
1.7.1	What's new and what changed from 1.7.17 to 1.7.18 . . . . .	7
1.7.2	What's new and what changed from 1.7.16 to 1.7.17 . . . . .	7
1.7.3	What's new and what changed from 1.7.15 to 1.7.16 . . . . .	7
1.7.4	What's new and what changed from 1.7.14 to 1.7.15 . . . . .	7
1.7.5	What's new and what changed from 1.7.13 to 1.7.14 . . . . .	7
1.7.6	What's new and what changed from 1.7.12 to 1.7.13 . . . . .	8
1.7.7	What's new and what changed from 1.7.11 to 1.7.12 . . . . .	8
1.7.8	What's new and what changed from 1.7.10 to 1.7.11 . . . . .	8
1.7.9	What's new and what changed from 1.7.9 to 1.7.10 . . . . .	8
1.7.10	What's new and what changed from 1.7.8 to 1.7.9 . . . . .	9
1.7.11	What's new and what changed from 1.7.7 to 1.7.8 . . . . .	9
1.7.12	What's new and what changed from 1.7.6 to 1.7.7 . . . . .	10

1.7.13	What's new and what changed from 1.7.5 to 1.7.6 . . . . .	10
1.7.14	What's new and what changed from 1.7.3 to 1.7.5 . . . . .	10
1.7.15	What's new and what changed from 1.7.2 to 1.7.3 . . . . .	10
1.7.16	What's new and what changed from 1.7.1 to 1.7.2 . . . . .	11
1.7.17	OS related changes . . . . .	12
1.7.18	File Access related changes . . . . .	12
1.7.19	Network related changes . . . . .	13
1.7.20	Device related changes . . . . .	14
1.7.21	Other POSIX related changes . . . . .	14
1.7.22	Security related changes . . . . .	14
1.7.23	Miscellaneous . . . . .	15

## **2 Setting Up Cygwin** **16**

2.1	Internet Setup . . . . .	16
2.1.1	Download Source . . . . .	16
2.1.2	Selecting an Install Directory . . . . .	16
2.1.3	Local Package Directory . . . . .	17
2.1.4	Connection Method . . . . .	17
2.1.5	Choosing Mirrors . . . . .	17
2.1.6	Choosing Packages . . . . .	17
2.1.7	Download and Installation Progress . . . . .	18
2.1.8	Shortcuts . . . . .	18
2.1.9	Post-Install Scripts . . . . .	18
2.1.10	Troubleshooting . . . . .	18
2.2	Environment Variables . . . . .	18
2.2.1	Overview . . . . .	18
2.2.2	Restricted Win32 environment . . . . .	19
2.3	Changing Cygwin's Maximum Memory . . . . .	19
2.4	Internationalization . . . . .	20
2.4.1	Overview . . . . .	20
2.4.2	How to set the locale . . . . .	21
2.4.3	The Windows Console character set . . . . .	22
2.4.4	Potential Problems when using Locales . . . . .	23
2.4.5	List of supported character sets . . . . .	23
2.5	Customizing bash . . . . .	25

---

<b>3</b>	<b>Using Cygwin</b>	<b>26</b>
3.1	Mapping path names	26
3.1.1	Introduction	26
3.1.2	The Cygwin Mount Table	26
3.1.3	UNC paths	29
3.1.4	The cygdrive path prefix	29
3.1.5	Using native Win32 paths	30
3.1.6	Using the Win32 file API in Cygwin applications	30
3.1.7	Additional Path-related Information	31
3.2	Text and Binary modes	31
3.2.1	The Issue	31
3.2.2	The default Cygwin behavior	32
3.2.3	Binary or text?	32
3.2.4	Programming	32
3.3	File permissions	33
3.4	Special filenames	33
3.4.1	Special files in /etc	33
3.4.2	Invalid filenames	34
3.4.3	Forbidden characters in filenames	34
3.4.4	Filenames with unusual (foreign) characters	34
3.4.5	Case sensitive filenames	35
3.4.6	POSIX devices	36
3.4.7	The .exe extension	38
3.4.8	The /proc filesystem	38
3.4.9	The /proc/registry filesystem	38
3.4.10	The @pathnames	39
3.5	The CYGWIN environment variable	39
3.5.1	Implemented options	39
3.5.2	Obsolete options	40
3.6	Using Windows security in Cygwin	41
3.6.1	Overview	41
3.6.2	File permissions	44
3.6.3	Special values of user and group ids	45
3.6.4	The POSIX permission mapping leak	46
3.6.5	Switching the user context	47
3.6.6	Switching the user context with password authentication	47
3.6.7	Switching the user context without password, Method 1: Create a token from scratch	48
3.6.8	Switching the user context without password, Method 2: LSA authentication package	49
3.6.9	Switching the user context without password, Method 3: With password	49

---

---

3.6.10	Switching the user context, how does it all fit together? . . . . .	50
3.7	Cygserver . . . . .	50
3.7.1	What is Cygserver? . . . . .	50
3.7.2	Cygserver command line options . . . . .	51
3.7.3	How to start Cygserver . . . . .	52
3.7.4	The Cygserver configuration file . . . . .	52
3.8	Using Cygwin effectively with Windows . . . . .	53
3.8.1	Pathnames . . . . .	53
3.8.2	Cygwin and Windows Networking . . . . .	53
3.8.3	The cygutils package . . . . .	53
3.8.4	Creating shortcuts with cygutils . . . . .	54
3.8.5	Printing with cygutils . . . . .	54
<b>4</b>	<b>Programming with Cygwin</b>	<b>55</b>
4.1	Using GCC with Cygwin . . . . .	55
4.1.1	Console Mode Applications . . . . .	55
4.1.2	GUI Mode Applications . . . . .	55
4.2	Debugging Cygwin Programs . . . . .	57
4.3	Building and Using DLLs . . . . .	58
4.3.1	Building DLLs . . . . .	58
4.3.2	Linking Against DLLs . . . . .	59
4.4	Defining Windows Resources . . . . .	60

---

# Chapter 1

## Cygwin Overview

### 1.1 What is it?

Cygwin is a Linux-like environment for Windows. It consists of a DLL (`cygwin1.dll`), which acts as an emulation layer providing substantial **POSIX** (Portable Operating System Interface) system call functionality, and a collection of tools, which provide a Linux look and feel. The Cygwin DLL works with all x86 and AMD64 versions of Windows NT since Windows 2000. The API follows the **Single Unix Specification** as much as possible, and then Linux practice. The major differences between Cygwin and Linux is the C library (`newlib` instead of `glibc`).

With Cygwin installed, users have access to many standard UNIX utilities. They can be used from one of the provided shells such as **bash** or from the Windows Command Prompt. Additionally, 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 distribution).

### 1.2 Quick Start Guide for those more experienced with Windows

If you are new to the world of UNIX, you may find it difficult to understand at first. This guide is not meant to be comprehensive, so we recommend that you use the many available Internet resources to become acquainted with UNIX basics (search for "UNIX basics" or "UNIX tutorial").

To install a basic Cygwin environment, run the **setup.exe** program and click **Next** at each page. The default settings are correct for most users. If you want to know more about what each option means, see Section 2.1. Use **setup.exe** any time you want to update or install a Cygwin package. If you are installing Cygwin for a specific purpose, use it to install the tools that you need. For example, if you want to compile C++ programs, you need the `gcc-g++` package and probably a text editor like `nano`. When running **setup.exe**, clicking on categories and packages in the package installation screen will provide you with the ability to control what is installed or updated.

Another option is to install everything by clicking on the `Default` field next to the `All` category. However, be advised that this will download and install several hundreds of megabytes of software to your computer. The best plan is probably to click on individual categories and install either entire categories or packages from the categories themselves. After installation, you can find Cygwin-specific documentation in the `/usr/share/doc/Cygwin/` directory.

Developers coming from a Windows background will be able to write console or GUI executables that rely on the Microsoft Win32 API instead of Cygwin using the `mingw32` or `mingw64` cross-compiler toolchains. The **-shared** option to GCC allows to write Windows Dynamically Linked Libraries (DLLs). The resource compiler **windres** is also provided.

### 1.3 Quick Start Guide for those more experienced with UNIX

If you are an experienced UNIX user who misses a powerful command-line environment, you will enjoy Cygwin. 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 will find that the Cygwin library provides an easy way to port many UNIX packages, with only minimal source code changes.

Note that there are some workarounds that cause Cygwin to behave differently than most UNIX-like operating systems; these are described in more detail in Section 3.8.

Use the graphical command **setup.exe** any time you want to update or install a Cygwin package. This program must be run manually every time you want to check for updated packages since Cygwin does not currently include a mechanism for automatically detecting package updates.

By default, **setup.exe** only installs a minimal subset of packages. Add any other packages by clicking on the + next to the Category name and selecting the package from the displayed list. You may search for specific tools by using the [Setup Package Search](#) at the Cygwin web site.

Another option is to install everything by clicking on the `Default` field next to the `All` category. However, be advised that this will download and install several hundreds of megabytes of software to your computer. The best plan is probably to click on individual categories and install either entire categories or packages from the categories themselves. After installation, you can find Cygwin-specific documentation in the `/usr/share/doc/Cygwin/` directory.

For more information about what each option in **setup.exe** means, see Section 2.1.

## 1.4 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 Red Hat and placed under the [GNU General Public License](#) (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 for more information on how the GNU GPL 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 go to <http://www.redhat.com/services/custom/cygwin/>. Customers of the native Win32 GNUPro should feel free to submit bug reports and ask questions through Red Hat channels. All other questions should be sent to the project mailing list [cygwin@cygwin.com](mailto:cygwin@cygwin.com).

## 1.5 A brief history of the Cygwin project

---

### Note

A historical look into the first years of Cygwin development is Geoffrey J. Noer's 1998 paper, "Cygwin32: A Free Win32 Porting Layer for UNIX® Applications" which can be found at the [2nd USENIX Windows NT Symposium Online Proceedings](#).

---

Cygwin began development in 1995 at Cygnus Solutions (now part of Red Hat, Inc.). The first thing done was to enhance the development tools (**gcc**, **gdb**, **gas**, etc.) so that they could generate and 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 Windows-hosted 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. Many 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 since Cygwin provided a UNIX-like API. Running `bash` with the development tools and user tools in place, Windows 9x and NT looked like a flavor of UNIX from the perspective of the GNU configure mechanism. Self hosting was achieved as of the beta 17.1 release in October 1996.

The entire Cygwin toolset was available as a monolithic install. In April 2000, the project announced a [New Cygwin Net Release](#) which provided the native non-Cygwin Win32 program **setup.exe** to install and upgrade each package separately. Since then, the Cygwin DLL and **setup.exe** have seen continuous development.

---



The latest major improvement in this development is the 1.7 release in 2009, which dropped Windows 95/98/Me support in favor of using Windows NT features more extensively. It adds a lot of new features like case-sensitive filenames, NFS interoperability, IPv6 support and much more.

## 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 and global synchronization objects that other processes using separate instances of the DLL can access. This is used to keep track of open file descriptors and to assist fork and exec, among other purposes. 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 as a standard DLL in the Win32 subsystem. Under the hood it's using the Win32 API, as well as the native NT API, where appropriate.

---

**Note**

Some restrictions apply for calls to the Win32 API. For details, see Section 2.2.2, as well as Section 3.1.6.

---

The native NT API is used mainly for speed, as well as to access NT capabilities which are useful to implement certain POSIX features, but are hidden to the Win32 API.

Due to some restrictions in Windows, it's not always possible to strictly adhere to existing UNIX standards like POSIX.1. Fortunately these are mostly corner cases.

Note that many of the things that Cygwin does to provide POSIX compatibility do not mesh well with the native Windows API. If you mix POSIX calls with Windows calls in your program it is possible that you will see uneven results. In particular, Cygwin signals will not work with Windows functions which block and Windows functions which accept filenames may be confused by Cygwin's support for long filenames.

### 1.6.2 Permissions and Security

Windows NT includes a sophisticated security model based on Access Control Lists (ACLs). Cygwin maps Win32 file ownership and permissions to ACLs by default, on file systems supporting them (usually NTFS). Solaris style ACLs and accompanying function calls are also supported. 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.

Users with Administrator rights are permitted to chown files. With version 1.1.3 Cygwin introduced a mechanism for setting real and effective UIDs. This is described in Section 3.6. As of version 1.5.13, the Cygwin developers are not aware of any feature in the Cygwin DLL that would allow users to gain privileges or to access objects to which they have no rights under Windows. However there is no guarantee that Cygwin is as secure as the Windows it runs on. Cygwin processes share some variables and are thus easier targets of denial of service type of attacks.

### 1.6.3 File Access

Cygwin supports both POSIX- and Win32-style paths, using either forward or back slashes as the directory delimiter. Paths coming into the DLL are translated from POSIX to native NT as needed. From the application perspective, the file system is a POSIX-compliant one. The implementation details are safely hidden in the Cygwin DLL. UNC pathnames (starting with two slashes) are supported for network paths.

Since version 1.7.0, the layout of this POSIX view of the Windows file system space is stored in the `/etc/fstab` file. Actually, there is a system-wide `/etc/fstab` file as well as a user-specific `fstab` file `/etc/fstab.d/${USER}`.

---

At startup the DLL has to find out where it can find the `/etc/fstab` file. The mechanism used for this is simple. First it retrieves it's own path, for instance `C:\Cygwin\bin\cygwin1.dll`. From there it deduces that the root path is `C:\Cygwin`. So it looks for the `fstab` file in `C:\Cygwin\etc\fstab`. The layout of this file is very similar to the layout of the `fstab` file on Linux. Just instead of block devices, the mount points point to Win32 paths. An installation with **setup.exe** installs a `fstab` file by default, which can easily be changed using the editor of your choice.

The `fstab` file allows mounting arbitrary Win32 paths into the POSIX file system space. A special case is the so-called `cygdrive` prefix. It's the path under which every available drive in the system is mounted under its drive letter. The default value is `/cygdrive`, so you can access the drives as `/cygdrive/c`, `/cygdrive/d`, etc... The `cygdrive` prefix can be set to some other value (`/mnt` for instance) in the `fstab` file(s).

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 applications handle filenames in a case preserving, but case insensitive manner. Cygwin supports case sensitivity on file systems supporting that. Since Windows XP, the OS only supports case sensitivity when a specific registry value is changed. Therefore, case sensitivity is not usually the default.

Symbolic links are not present and supported on Windows up to and including Windows Server 2003 R2. Native symlinks are available starting with Windows Vista. Due to their strange implementation, however, they are not useful in a POSIX emulation layer. Cygwin recognizes native symlinks, but does not create them.

Symbolic links are potentially created in two different ways. The file style symlinks are files containing a magic cookie followed by the path to which the link points. They are marked with the System DOS attribute so that only files with that attribute have to be read to determine whether or not the file is a symbolic link. The shortcut style symlinks are Windows shortcut files with a special header and the Readonly DOS attribute set. The advantage of file symlinks is speed, the advantage of shortcut symlinks is the fact that they can be utilized by non-Cygwin Win32 tools as well.

Starting with Cygwin 1.7, symbolic links are using UTF-16 to encode the filename of the target file, to better support internationalization. Symlinks created by older Cygwin releases can be read just fine. However, you could run into problems with them if you're now using another character set than the one you used when creating these symlinks (see Section 2.4.4. Please note that this new UTF-16 style of symlinks is not compatible with older Cygwin release, which can't read the target filename correctly.

Hard links are fully supported on NTFS and NFS file systems. On FAT and other file systems which don't support hardlinks, the call returns with an error, just like on other POSIX systems.

On file systems which don't support unique persistent file IDs (FAT, older Samba shares) 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.

Cygwin 1.7 and later supports Extended Attributes (EAs) via the linux-specific function calls `getxattr`, `setxattr`, `listxattr`, and `removexattr`. All EAs on Samba or NTFS are treated as user EAs, so, if the name of an EA is "foo" from the Windows perspective, it's transformed into "user.foo" within Cygwin. This allows Linux-compatible EA operations and keeps tools like **attr**, or **setfattr** happy.

`chroot` is supported since Cygwin 1.1.3. However, `chroot` is not a concept known by Windows. This implies some serious restrictions. First of all, the `chroot` call isn't a privileged call. Any user may call it. Second, the `chroot` environment isn't safe against native windows processes. Given that, `chroot` in Cygwin is only a hack which pretends security where there is none. For that reason the usage of `chroot` is discouraged.

#### 1.6.4 Text Mode vs. Binary Mode

It is often important that files created by native Windows applications be interoperable with Cygwin applications. For example, a file created by a native Windows text editor should be readable by a Cygwin application, and vice versa.

Unfortunately, UNIX and Win32 have different end-of-line conventions in text files. A UNIX text file will have a single newline character (LF) whereas a Win32 text file will instead use a two character sequence (CR+LF). Consequently, the two character sequence must be translated on the fly by Cygwin into a single character newline when reading in text mode.

This solution addresses the newline interoperability concern at the expense of violating the POSIX requirement that text and binary mode be identical. Consequently, processes that attempt to lseek through text files can no longer rely on the number of

bytes read to be an accurate indicator of position within the file. For this reason, Cygwin allows you to choose the mode in which a file is read in several ways.

### 1.6.5 ANSI C Library

We chose to include Red Hat's 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. Other functions, which are not supported by newlib have been added to the Cygwin sources using BSD implementations as much as possible.

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.

### 1.6.6 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.

#### 1.6.6.1 Problems with process creation

The semantics of `fork` require that a forked child process have *exactly* the same address space layout as its parent. However, Windows provides no native support for cloning address space between processes and several features actively undermine a reliable `fork` implementation. Three issues are especially prevalent:

- **DLL base address collisions.** Unlike \*nix shared libraries, which use "position-independent code", Windows shared libraries assume a fixed base address. Whenever the hard-wired address ranges of two DLLs collide (which occurs quite often), the Windows loader must "rebase" one of them to a different address. However, it may not resolve collisions consistently, and may rebase a different dll and/or move it to a different address every time. Cygwin can usually compensate for this effect when it involves libraries opened dynamically, but collisions among statically-linked dlls (dependencies known at compile time) are resolved before `cygwin1.dll` initializes and cannot be fixed afterward. This problem can only be solved by removing the base address conflicts which cause the problem, usually using the `rebaseall` tool.

- Address space layout randomization (ASLR). Starting with Vista, Windows implements ASLR, which means that thread stacks, heap, memory-mapped files, and statically-linked dlls are placed at different (random) locations in each process. This behaviour interferes with a proper `fork`, and if an unmovable object (process heap or system dll) ends up at the wrong location, Cygwin can do nothing to compensate (though it will retry a few times automatically).
- DLL injection by **BLODA**. Badly-behaved applications which inject dlls into other processes often manage to clobber important sections of the child's address space, leading to base address collisions which rebasing cannot fix. The only way to resolve this problem is to remove (usually uninstall) the offending app. See Section 3.5.1 for the `detect_bloda` option, which may be able to identify the BLODA.

In summary, current Windows implementations make it impossible to implement a perfectly reliable fork, and occasional fork failures are inevitable.

### 1.6.7 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.

### 1.6.8 Sockets

Socket-related calls in Cygwin basically call the functions by the same name in Winsock, Microsoft's implementation of Berkeley sockets, but with lots of tweaks. All sockets are non-blocking under the hood to allow to interrupt blocking calls by POSIX signals. Additional bookkeeping is necessary to implement correct socket sharing POSIX semantics and especially for the `select` call. Some socket-related functions are not implemented at all in Winsock, as, for example, `socketpair`. Starting with Windows Vista, Microsoft removed the legacy calls `rcmd(3)`, `rexec(3)` and `rresvport(3)`. Recent versions of Cygwin now implement all these calls internally.

An especially troublesome feature of Winsock is that it must be initialized before the first socket function is called. As a result, Cygwin has to perform this initialization on the fly, as soon as the first socket-related function is called by the application. In order to support sockets across fork calls, child processes initialize Winsock if any inherited file descriptor is a socket.

AF\_UNIX (AF\_LOCAL) sockets are not available in Winsock. They are implemented in Cygwin by using local AF\_INET sockets instead. This is completely transparent to the application. Cygwin's implementation also supports the `getpeereid` BSD extension. However, Cygwin does not yet support descriptor passing.

IPv6 is supported beginning with Cygwin release 1.7.0. This support is dependent, however, on the availability of the Windows IPv6 stack. The IPv6 stack was "experimental", i.e. not feature complete in Windows 2003 and earlier. Full IPv6 support became available starting with Windows Vista and Windows Server 2008. Cygwin does not depend on the underlying OS for the (newly implemented) `getaddrinfo` and `getnameinfo` functions. Cygwin 1.7.0 adds replacement functions which implement the full functionality for IPv4.

### 1.6.9 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.

## 1.7 What's new and what changed in Cygwin 1.7

### 1.7.1 What's new and what changed from 1.7.17 to 1.7.18

- Added Windows console cursor appearance support.
  - Show/Hide Cursor mode (DECTCEM): `"ESC[?25h" / "ESC[?25l"`
  - Set cursor style (DECSCUSR): `"ESC[n q"` (note the space before the q); where n is 0, 1, 2 for block cursor, 3, 4 for underline cursor (all disregarding blinking mode), or > 4 to set the cursor height to a percentage of the cell height.
- For performance reasons, Cygwin does not try to create sparse files automatically anymore, unless you use the new "sparse" mount option.
- New API: `cfsetspeed`.

### 1.7.2 What's new and what changed from 1.7.16 to 1.7.17

- Support the "e" flag to `fopen(3)`. This is a Glibc extension which allows to `fopen` the file with the `O_CLOEXEC` flag set.
- Support the "x" flag to `fopen(3)`. This is a Glibc/C11 extension which allows to open the file with the `O_EXCL` flag set.

### 1.7.3 What's new and what changed from 1.7.15 to 1.7.16

- New API: `getmntent_r`, `memrchr`.
- Recognize ReFS filesystem.

### 1.7.4 What's new and what changed from 1.7.14 to 1.7.15

- `CYGWIN=pipe_byte` option now forces the opening of pipes in byte mode rather than message mode.

### 1.7.5 What's new and what changed from 1.7.13 to 1.7.14

- Add mouse reporting modes 1005, 1006 and 1015 to console window.
-

### 1.7.6 What's new and what changed from 1.7.12 to 1.7.13

- `mkpasswd` and `mkgroup` now try to print an entry for the TrustedInstaller account existing since Windows Vista/Server 2008.
- Terminal typeahead when switching from canonical to non-canonical mode is now properly flushed.

### 1.7.7 What's new and what changed from 1.7.11 to 1.7.12

- Cygwin now automatically populates the `/dev` directory with all existing POSIX devices.
- Add virtual `/proc/PID/mountinfo` file.
- `flock` now additionally supports the following scenario, which requires to propagate locks to the parent process:

```
(
  flock -n 9 || exit 1
  # ... commands executed under lock ...
} 9>/var/lock/mylockfile
```

Only propagation to the direct parent process is supported so far, not to grand parents or sibling processes.

- Add a "detect\_bloda" setting for the CYGWIN environment variable to help finding potential BLODAs.

### 1.7.8 What's new and what changed from 1.7.10 to 1.7.11

- New `pldd` command for listing DLLs loaded by a process.
- New API: `scandirat`.
- Change the way remote shares mapped to drive letters are recognized when creating the `cygdrive` directory. If Windows claims the drive is unavailable, don't show it in the `cygdrive` directory listing.
- Raise default stacksize of pthreads from 512K to 1 Meg. It can still be changed using the `pthread_attr_setstacksize` call.

### 1.7.9 What's new and what changed from 1.7.9 to 1.7.10

- Drop support for Windows NT4.
- The CYGWIN environment variable options "envcache", "strip\_title", "title", "tty", and "upcaseenv" have been removed.
- If the executable (and the system) is large address aware, the application heap will be placed in the large memory area. The `peflags` tool from the `rebase` package can be used to set the large address awareness flag in the executable file header.
- The registry setting "heap\_chunk\_in\_mb" has been removed, in favor of a new per-executable setting in the executable file header which can be set using the `peflags` tool. See Section 2.3 for more information.
- The CYGWIN=tty mode using pipes to communicate with the console in a pseudo tty-like mode has been removed. Either just use the normal Windows console as is, or use a terminal application like **mintty**.
- New `getconf` command for querying `confstr(3)`, `pathconf(3)`, `sysconf(3)`, and `limits.h` configuration.
- New `tzset` utility to generate a POSIX-compatible TZ environment variable from the Windows timezone settings.
- The `passwd` command now allows an administrator to use the `-R` command for other user accounts: `passwd -R username`.
- Pthread spinlocks. New APIs: `pthread_spin_destroy`, `pthread_spin_init`, `pthread_spin_lock`, `pthread_spin_trylock`, `pthread_spin_unlock`.
- Pthread stack address management. New APIs: `pthread_attr_getstack`, `pthread_attr_getstackaddr`, `pthread_attr_getguardsize`, `pthread_attr_setstack`, `pthread_attr_setstackaddr`, `pthread_attr_setguardsize`, `pthread_getattr_np`.
- POSIX Clock Selection option. New APIs: `clock_nanosleep`, `pthread_condattr_getclock`, `pthread_condattr_setclock`.

- `clock_gettime(3)` and `clock_getres(3)` accept per-process and per-thread CPU-time clocks, including `CLOCK_PROCESS_CPUTIME_ID` and `CLOCK_THREAD_CPUTIME_ID`. New APIs: `clock_getcpuclockid`, `pthread_getcpuclockid`.
- GNU/glibc `error.h` error reporting functions. New APIs: `error`, `error_at_line`. New exports: `error_message_count`, `error_one_per_line`, `error_print_progname`. Also, `perror` and `strerror_r` no longer clobber `strerror` storage.
- C99 `<tgmath.h>` type-generic macros.
- `/proc/loadavg` now shows the number of currently running processes and the total number of processes.
- Added `/proc/devices` and `/proc/misc`, which lists supported device types and their device numbers.
- Added `/proc/swaps`, which shows the location and size of Windows paging file(s).
- Added `/proc/sysvipc/msg`, `/proc/sysvipc/sem`, and `/proc/sysvipc/shm` which provide information about System V IPC message queues, semaphores, and shared memory.
- `/proc/version` now shows the username of whomever compiled the Cygwin DLL as well as the version of GCC used when compiling.
- `dlopen` now supports the Glibc-specific `RTLD_NODELETE` and `RTLD_NOOPEN` flags.
- The `printf(3)` and `wprintf(3)` families of functions now handle the `%m` conversion flag.
- Other new API: `clock_settime`, `__fpurge`, `getgrouplist`, `get_current_dir_name`, `getpt`, `ppoll`, `psiginfo`, `psignal`, `ptsname_r`, `sys_siglist`, `pthread_setschedprio`, `pthread_sigqueue`, `sysinfo`.

### 1.7.10 What's new and what changed from 1.7.8 to 1.7.9

- New API: `strchrnul`.

### 1.7.11 What's new and what changed from 1.7.7 to 1.7.8

- Drop support for Windows NT4 prior to Service Pack 4.
- Reinstantiate Cygwin's ability to delete an empty directory which is the current working directory of the same or another process. Same for any other empty directory which has been opened by the same or another process.
- Cygwin now ships the C standard library `fenv.h` header file, and implements the related APIs (including GNU/glibc extensions): `feclearexcept`, `fedisableexcept`, `feenableexcept`, `fegetenv`, `fegetexcept`, `fegetexceptflag`, `fegetprec`, `fegetround`, `fehldexcept`, `feraiseexcept`, `fesetenv`, `fesetexceptflag`, `fesetprec`, `fesetround`, `fetestexcept`, `feupdateenv`, and predefines both default and no-mask FP environments. See the [GNU C Library manual](#) for full details of this functionality.
- Support for the C99 complex functions, except for the "long double" implementations. New APIs: `cacos`, `cacosf`, `cacosh`, `cacoshf`, `carg`, `cargf`, `casin`, `casinf`, `casinh`, `casinhf`, `catan`, `catanf`, `catanh`, `catanhf`, `ccos`, `ccosf`, `ccosh`, `ccoshf`, `cexp`, `cexpf`, `cimag`, `cimagf`, `clog`, `clogf`, `conj`, `conjf`, `cpow`, `cpowf`, `cproj`, `cprojf`, `creal`, `crealf`, `csin`, `csinf`, `csinh`, `csinhf`, `csqrt`, `csqrtf`, `ctan`, `ctanf`, `ctanh`, `ctanhf`.
- Fix the width of "CJK Ambiguous Width" characters to 1 for singlebyte charsets and 2 for East Asian multibyte charsets. (For UTF-8, it remains dependent on the specified language, and the `"@cjkcnarrow"` locale modifier can still be used to force width 1.)
- The `strerror_r` interface now has two flavors; if `_GNU_SOURCE` is defined, it retains the previous behavior of returning `char *` (but the result is now guaranteed to be NUL-terminated); otherwise it now obeys POSIX semantics of returning `int`.
- `/proc/sys` now allows unfiltered access to the native NT namespace. Access restrictions still apply. Direct device access via `/proc/sys` is not yet supported. File system access via block devices works. For instance (note the trailing slash!)

```
bash$ cd /proc/sys/Device/HarddiskVolumeShadowCopy1/
```

- Other new APIs: `llround`, `llroundf`, `madvise`, `pthread_yield`. Export `program_invocation_name`, `program_invocation_short_name`. Support `TIOCGPGRP`, `TIOCSGRP` ioctls.



### 1.7.12 What's new and what changed from 1.7.6 to 1.7.7

- Partially revert the 1.7.6 change to set the Win32 current working directory (CWD) always to an invalid directory, since it breaks backward compatibility too much. The Cygwin CWD and the Win32 CWD are now kept in sync again, unless the Cygwin CWD is not usable as Win32 CWD. See the reworked Section 3.1.6 for details.
- Make sure to follow the Microsoft security advisory concerning DLL hijacking. See the [Microsoft Security Advisory \(2269637\) "Insecure Library Loading Could Allow Remote Code Execution"](#) for details.
- Allow to link against `-lbinmode` instead of `/lib/binmode.o`. Same for `-ltextmode`, `-ltextreadmode` and `-lautomode`. See Section 3.2.4 for details.

### 1.7.13 What's new and what changed from 1.7.5 to 1.7.6

- Add new mount options "dos" and "ihash" to allow overriding Cygwin default behaviour on broken filesystems not recognized by Cygwin.
- Add new mount option "bind" to allow remounting parts of the POSIX file hierarchy somewhere else.
- Ttys and ptys are handled as securable objects using file-like permissions and owner/group information. **chmod** and **chown** now work on ttys/ptys. A new mechanism is used to propagate pty handles safely to other processes, which does not require to use Cygserver.
- Pass on `coresize` settings made with `setrlimit(2)`. This allows shells to disable creating stackdump files in child processes via

```
ulimit -c 0
```

in bash or

```
limit coredumpsize 0
```

in tcsh.

- Locale categories contain all localization strings additionally as wide-char strings. `locale(1)` prints these values just as on Linux. `nl_langinfo(3)` allows to fetch them.
- New interfaces `mkostemp(3)` and `mkostemps(3)` are added.
- New virtual file `/proc/filesystems`.
- `clock_gettime(3)` and `clock_getres(3)` accept `CLOCK_MONOTONIC`.
- DEPRECATED with 1.7.7: Cygwin handles the current working directory entirely on its own. The Win32 current working directory is set to an invalid path to be out of the way. [...]

### 1.7.14 What's new and what changed from 1.7.3 to 1.7.5

- Support for DEC Backarrow Key Mode escape sequences (ESC [ ? 67 h, ESC [ ? 67 l) in Windows console.

### 1.7.15 What's new and what changed from 1.7.2 to 1.7.3

- Support for GB2312/EUC-CN. These charsets are implemented as aliases to GBK. GB2312 is now the default charset name for the locales `zh_CN` and `zh_SG`, just as on Linux.
- Modification and access timestamps of devices reflect the current time.



### 1.7.16 What's new and what changed from 1.7.1 to 1.7.2

- Localization support has been much improved.
    - Cygwin now handles locales using the underlying Windows locale support. The locale must exist in Windows to be recognized. Locale aliases from the file `/usr/share/locale/locale.alias` are also allowed, as long as their replacement is supported by the underlying Windows.
    - New tool "locale" to fetch locale information and default locales based on the Windows default settings as well as lists of all supported locales and character sets.
    - Default charset for locales without explicit charset is now chosen from a list of Linux-compatible charsets.  
For instance: `en_US` -> `ISO-8859-1`, `ja_JP` -> `EUC-JP`, `zh_TW` -> `Big5`.
    - Added support for the charsets `GEORGIAN-PS`, `PT154`, and `TIS-620`.
    - Support for the various locale modifiers to switch charsets as on Linux.
    - Default charset in the "C" or "POSIX" locale has been changed back from UTF-8 to ASCII, to avoid problems with applications expecting a singlebyte charset in the "C"/"POSIX" locale. Still use UTF-8 internally for filename conversion in this case.
    - `LC_COLLATE`, `LC_MONETARY`, `LC_NUMERIC`, and `LC_TIME` localization is enabled via Windows locale support. `LC_MESSAGES` is enabled via a big table with localized strings.
    - `fnmatch(3)`, `regcomp(3)`, `regex(3)` calls are now multibyte-aware.
    - `printf(3)`, `wprintf(3)` families of functions now handle the grouping flag, the apostrophe ' , per POSIX-1.2008. The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, `%G`) will be formatted with thousands' grouping characters.
    - `strftime(3)`, `wcsftime(3)`, and `strptime(3)` now handle the E and O format modifiers to print/scan alternative date and time representations or to use alternative digits in locales which support this. Additionally these functions now also support the padding modifiers '0' and '+', as well as a field width per POSIX-1.2008.
    - New `strfmon(3)` call.
  - Support `open(2)` flags `O_CLOEXEC` and `O_TTY_INIT` flags. Support `fcntl` flag `F_DUPFD_CLOEXEC`. Support socket flags `SOCK_CLOEXEC` and `SOCK_NONBLOCK`. Add new Linux-compatible API calls `accept4(2)`, `dup3(2)`, and `pipe2(2)`. Support the signal `SIGPWR`.
  - Enhanced Windows console support.
    - The console's backspace keycode can be changed using 'stty erase'.
    - Function keys send distinguished escape sequences compatible with `rxvt`. Keypad keys send distinguished escape sequences, `xterm`-style.
    - Support of combining Alt and AltGr modifiers in console window (compatible with `xterm` and `mintty`), so that e.g. Alt-@ sends ESC @ also on keyboards where @ is mapped to an AltGr combination.
    - Report mouse wheel scroll events in mouse reporting mode 1000 (note: this doesn't seem to work on all systems, assumedly due to driver interworking issues). Add mouse reporting mode 1002 to report mouse drag movement. Add mouse reporting mode 1003 to report any mouse movement. Add focus event reporting (mode 1004), compatible with `xterm` and `mintty`.
    - Add escape sequences for not bold (22), not invisible (28), not blinking (25) (compatible with `xterm` and `mintty`).
    - Support VT100 line drawing graphics mode in console window (compatible with `xterm` and `mintty`).
  - Handle native DOS paths always as if mounted with "posix=0,noacl".
  - Handle UNC paths starting with slashes identical to /cygdrive paths. In other words, use the /cygdrive mount flags for these paths as well.
  - Recognize NWFS filesystem and workaround broken OS call.
  - New support for eXtensible Data Record (XDR) encoding and decoding, as defined by RFCs 1014, 1832, and 4506. The XDR protocol and functions are useful for cross-platform data exchange, and are commonly used as the core data interchange format for Remote Procedure Call (RPC) and NFS.
-

### 1.7.17 OS related changes

- Windows 95, 98 and Me are not supported anymore. The new Cygwin 1.7 DLL will not run on any of these systems.
- Add support for Windows 7 and Windows Server 2008 R2.

### 1.7.18 File Access related changes

- Mount points are no longer stored in the registry. Use `/etc/fstab` and `/etc/fstab.d/$USER` instead. Mount points created with `mount(1)` are only local to the current session and disappear when the last Cygwin process in the session exits.
- Cygwin creates the mount points for `/`, `/usr/bin`, and `/usr/lib` automatically from its own position on the disk. They don't have to be specified in `/etc/fstab`.
- If a filename cannot be represented in the current character set, the character will be converted to a sequence `Ctrl-X + UTF-8` representation of the character. This allows to access all files, even those not having a valid representation of their filename in the current character set. To always have a valid string, use the UTF-8 charset by setting the environment variable `$LANG`, `$LC_ALL`, or `$LC_CTYPE` to a valid POSIX value, such as `"en_US.UTF-8"`.
- `PATH_MAX` is now 4096. Internally, path names can be as long as the underlying OS can handle (32K).
- `struct dirent` now supports `d_type`, filled out with `DT_REG` or `DT_DIR`. All other file types return as `DT_UNKNOWN` for performance reasons.
- The CYGWIN environment variable options `"ntsec"` and `"smbntsec"` have been replaced by the per-mount option `"acl"/"noacl"`.
- The CYGWIN environment variable option `"ntea"` has been removed without substitute.
- The CYGWIN environment variable option `"check_case"` has been removed in favor of real case-sensitivity on file systems supporting it.
- Creating filenames with special DOS characters `'"', '!', ':', '<', '>', '|'` is supported.
- Creating files with special DOS device filename components (`"aux"`, `"nul"`, `"prn"`) is supported.
- File names are case sensitive if the OS and the underlying file system supports it. Works on NTFS and NFS. Does not work on FAT and Samba shares. Requires to change a registry key (see the User's Guide). Can be switched off on a per-mount basis.
- Due to the above changes, managed mounts have been removed.
- Incoming DOS paths are always handled case-insensitive and get no POSIX permission, as if they are mounted with `noacl,posix=0` mount flags.
- `unlink(2)` and `rmdir(2)` try very hard to remove files/directories even if they are currently accessed or locked. This is done by utilizing the hidden recycle bin directories and marking the files for deletion.
- `rename(2)` rewritten to be more POSIX conformant.
- `access(2)` now performs checks using the real user ID, as required by POSIX; the old behavior of querying based on effective user ID is available through the new `faccessat(2)` and `euidaccess(2)` APIs.
- Add `st_birthtim` member to `struct stat`.
- File locking is now advisory, not mandatory anymore. The `fcntl(2)` and the new `lockf(2)` APIs create and maintain locks with POSIX semantics, the `flock(2)` API creates and maintains locks with BSD semantics. POSIX and BSD locks are independent of each other.
- Implement atomic `O_APPEND` mode.
- New `open(2)` flags `O_DIRECTORY`, `O_EXEC` and `O_SEARCH`.
- Make the "plain file with SYSTEM attribute set" style symlink default again when creating symlinks. Only create Windows shortcut style symlinks if `CYGWIN=winsymlinks` is set in the environment.

- Symlinks now use UTF-16 encoding for the target filename for better internationalization support. Cygwin 1.7 can read all old style symlinks, but the new style is not compatible with older Cygwin releases.
- Handle NTFS native symlinks available since Vista/2008 as symlinks (but don't create Vista/2008 symlinks due to unfortunate OS restrictions).
- Recognize NFS shares and handle them using native mechanisms. Recognize and create real symlinks on NFS shares. Get correct stat(2) information and set real mode bits on open(2), mkdir(2) and chmod(2).
- Recognize MVFS and workaround problems manipulating metadata and handling DOS attributes.
- Recognize Netapp DataOnTap drives and fix inode number handling.
- Recognize Samba version beginning with Samba 3.0.28a using the new extended version information negotiated with the Samba developers.
- Stop faking hardlinks by copying the file on filesystems which don't support hardlinks natively (FAT, FAT32, etc.). Just return an error instead, just like Linux.
- List servers of all accessible domains and workgroups in // instead of just the servers in the own domain/workgroup.
- Support Linux-like extended attributes ([fl]getattr, [fl]listattr, [fl]setattr, [fl]removexattr).
- New file conversion API for conversion from Win32 to POSIX path and vice versa (cygwin\_conv\_path, cygwin\_create\_path, cygwin\_conv\_path\_list).
- New openat family of functions: openat, faccessat, fchmodat, fchownat, fstatat, futimesat, linkat, mkdirat, mkfifoat, mknodat, readlinkat, renameat, symlinkat, unlinkat.
- Other new APIs: posix\_fadvise, posix\_fallocate, funopen, fopencookie, open\_memstream, open\_wmemstream, fmemopen, fdopendir, fpurge, mkstemp, eaccess, euidaccess, canonicalize\_file\_name, fexecve, execvpe.

### 1.7.19 Network related changes

- New implementation for blocking sockets and select on sockets which is supposed to allow POSIX-compatible sharing of sockets between threads and processes.
- send/sendto/sendmsg now send data in 64K chunks to circumvent an internal buffer problem in WinSock (KB 201213).
- New send/recv option MSG\_DONTWAIT.
- IPv6 support. New APIs getaddrinfo, getnameinfo, freeaddrinfo, gai\_strerror, in6addr\_any, in6addr\_loopback. On IPv6-less systems, replacement functions are available for IPv4. On systems with IPv6 enabled, the underlying WinSock functions are used. While I tried hard to get the functionality as POSIXy as possible, keep in mind that a \*fully\* conformant implementation of getaddrinfo and other stuff is only available starting with Windows Vista/2008.
- Resolver functions (res\_init, res\_query, res\_search, res\_querydomain, res\_mkquery, res\_send, dn\_comp, dn\_expand) are now part of Cygwin. Applications don't have to link against minires anymore. Actually, this \*is\* the former libminires.a.
- rcmd is now implemented inside of Cygwin, instead of calling the WinSock function. This allows rsh(1) usage on Vista/2008 and later, which dropped this function from WinSock.
- Define multicast structures in netinet/in.h. Note that fully conformant multicast support is only available beginning with Vista/2008.
- Improve get\_ifconf. Redefine struct ifreq and subsequent datastructures to be able to keep more information. Support SIOCGIFINDEX, SIOCGIFDSTADDR and the Cygwin specific SIOCGIFFRNDLYNAM. Support real interface flags on systems supporting them.
- Other new APIs: bindresvport, bindresvport\_sa, gethostbyname2, iruserok\_sa, rcmd\_af, rresvport\_af, getifaddrs, freeifaddrs, if\_nametoindex, if\_indextoname, if\_nameindex, if\_freenameindex.
- Add /proc/net/if\_inet6.

### 1.7.20 Device related changes

- Reworked pipe implementation which uses overlapped IO to create more reliable interruptible pipes and fifos.
- The CYGWIN environment variable option "binmode" has been removed.
- Improved fifo handling by using native Windows named pipes.
- Detect when a stdin/stdout which looks like a pipe is really a tty. Among other things, this allows a debugged application to recognize that it is using the same tty as the debugger.
- Support UTF-8 in console window.
- In the console window the backspace key now emits DEL (0x7f) instead of BS (0x08), Alt-Backspace emits ESC-DEL (0x1b,0x7f) instead of DEL (0x7f), same as the Linux console and xterm. Control-Space now emits an ASCII NUL (0x0) character.
- Support up to 64 serial interfaces using /dev/ttyS0 - /dev/ttyS63.
- Support up to 128 raw disk drives /dev/sda - /dev/sddx.
- New API: cfmakeraw, get\_avphys\_pages, get\_nprocs, get\_nprocs\_conf, get\_phys\_pages, posix\_openpt.

### 1.7.21 Other POSIX related changes

- A lot of character sets are supported now via a call to setlocale(). The setting of the environment variables \$LANG, \$LC\_ALL or \$LC\_CTYPE will be used. For instance, setting \$LANG to "de\_DE.ISO-8859-15" before starting a Cygwin session will use the ISO-8859-15 character set in the entire session. The default locale in the absence of one of the aforementioned environment variables is "C.UTF-8".

The full list of supported character sets: "ASCII", "ISO-8859-x" with x in 1-16, except 12, "UTF-8", Windows codepages "CPxxx", with xxx in (437, 720, 737, 775, 850, 852, 855, 857, 858, 862, 866, 874, 1125, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258), "KOI8-R", "KOI8-U", "SJIS", "GBK", "eucJP", "eucKR", and "Big5".

- Allow multiple concurrent read locks per thread for pthread\_rwlock\_t.
- Implement pthread\_kill(thread, 0) as per POSIX.
- New API for POSIX IPC: Named semaphores: sem\_open, sem\_close, sem\_unlink. Message queues: mq\_open, mq\_getattr, mq\_setattr, mq\_notify, mq\_send, mq\_timedsend, mq\_receive, mq\_timedreceive, mq\_close, mq\_unlink. Shared memory: shm\_open, shm\_unlink.
- Only declare expected functions in <strings.h>, don't include <string.h> from here.
- Support for WCONTINUED, WIFCONTINUED() added to waitpid and wait4.
- New APIs: \_Exit, confstr, insque, remque, sys\_sigabbrev, posix\_madvise, posix\_memalign, reallocf, exp10, exp10f, pow10, pow10f, lrint, lrintf, rint, rintf, llrint, llrintf, llrintl, lrintl, rintl, mbsnrtowcs, strcasestr, stpcpy, stpncpy, wcpcpy, wcpncpy, wcsnlen, wcsnrtombs, wcsftime, westod, westof, westoimax, westok, westol, westoll, westoul, westoull, westoumax, wcsxfrm, wcscasecmp, wcsncasecmp, fgetwc, fgetws, fputwc, fputws, fwide, getwc, getwchar, putwc, putwchar, ungetwc, asprintf, dprintf, vasprintf, vdprintf, wprintf, fwprintf, swprintf, vwprintf, vfwprintf, vswprintf, wscanf, fwscanf, swscanf, vwscanf, vfwscanf, vswscanf.

### 1.7.22 Security related changes

- Getting a domain user's groups is hopefully more bulletproof now.
- Cygwin now comes with a real LSA authentication package. This must be manually installed by a privileged user using the /bin/cyglssa-config script. The advantages and disadvantages are noted in <http://cygwin.com/ml/cygwin-developers/2006-11/msg00000.html>

- Cygwin now allows storage and use of user passwords in a hidden area of the registry. This is tried first when Cygwin is called by privileged processes to switch the user context. This allows, for instance, ssh public key sessions with full network credentials to access shares on other machines.
- New options have been added to the mkpasswd and mkgroup tools to ease use in multi-machine and multi-domain environments. The existing options have a slightly changed behaviour.

### 1.7.23 Miscellaneous

- New ldd utility, similar to Linux.
- New link libraries libdl.a, libresolv.a, librt.a.
- Fallout from the long path names: If the current working directory is longer than 260 bytes, or if the current working directory is a virtual path (like /proc, /cygdrive, //server), don't call native Win32 programs since they don't understand these paths.
- On the first usage of a DOS path (C:\foo, \\foo\bar), the Cygwin DLL emits a scary warning that DOS paths shouldn't be used. This warning may be disabled via the new CYGWIN=nodosfilewarning setting.
- The CYGWIN environment variable option "server" has been removed. Cygwin automatically uses cygserver if it's available.
- Allow environment of arbitrary size instead of a maximum of 32K.
- Don't force uppercase environment when started from a non-Cygwin process. Except for certain Windows and POSIX variables which are always uppercased, preserve environment case. Switch back to old behaviour with the new CYGWIN=upcaseenv setting.
- Detect and report a missing DLL on process startup.
- Add /proc/registry32 and /proc/registry64 paths to access 32 bit and 64 bit registry on 64 bit systems.
- Add the ability to distinguish registry keys and registry values with the same name in the same registry subtree. The key is called "foo" and the value will be called "foo%val" in this case.
- Align /proc/cpuinfo more closely to Linux content.
- Add /proc/\$PID/mounts entries and a symlink /proc/mounts pointing to /proc/self/mounts as on Linux.
- Optimized strstr and memmem implementation.
- Remove backwards compatibility with old signal masks. (Some *very* old programs which use signal masks may no longer work correctly).
- Cygwin now exports wrapper functions for libstdc++ operators new and delete, to support the toolchain in implementing full C++ standards conformance when working with shared libraries.
- Different Cygwin installations in different paths can be run in parallel without knowing of each other. The path of the Cygwin DLL used in a process is a key used when creating IPC objects. So different Cygwin DLLs are running in different namespaces.
- Each Cygwin DLL stores its path and installation key in the registry. This allows troubleshooting of problems which could be a result of having multiple concurrent Cygwin installations.

## Chapter 2

# Setting Up Cygwin

### 2.1 Internet Setup

To install the Cygwin net release, go to <http://cygwin.com/> and click on "Install Cygwin Now!". This will download a GUI installer called **setup.exe** which can be run to download a complete cygwin installation via the internet. Follow the instructions on each screen to install Cygwin.

The **setup.exe** installer is designed to be easy for new users to understand while remaining flexible for the experienced. The volunteer development team is constantly working on **setup.exe**; before requesting a new feature, check the wishlist in the [CVS README](#). It may already be present in the CVS version!

Since the default value for each option is the logical choice for most installations, you can get a working minimal Cygwin environment installed by simply clicking the `Next` button at each page. The only exception to this is choosing a Cygwin mirror, which you can choose by experimenting with those listed at <http://cygwin.com/mirrors.html>. For more details about each of page of the **setup.exe** installation, read on below. Please note that this guide assumes that you have a basic understanding of Unix (or a Unix-like OS). If you are new to Unix, you will also want to make use of [other resources](#).

#### 2.1.1 Download Source

Cygwin uses packages to manage installing various software. When the default `Install from Internet` option is chosen, **setup.exe** creates a local directory to store the packages before actually installing the contents. `Download from Internet` performs only the first part (storing the packages locally), while `Install from Local Directory` performs only the second (installing the contents of the packages).

The `Download from Internet` option is mainly for creating a base Cygwin package tree on one computer for installation on several machines with `Install from Local Directory`; copy the entire local package tree to another machine with the directory tree intact. For example, you might create a `C:\cache\` directory and place **setup.exe** in it. Run **setup.exe** to `Install from Internet` or `Download from Internet`, then copy the whole `C:\cache\` to each machine and instead choose `Install from Local Directory`.

Though this provides some basic mirroring functionality, if you are managing a large Cygwin installation, to keep up to date we recommend using a mirroring tool such as **wget**. A helpful user on the Cygwin mailing list created a simple demonstration script to accomplish this; search the list for **mkcygwget** for ideas.

#### 2.1.2 Selecting an Install Directory

The `Root Directory` for Cygwin (default `C:\cygwin`) will become `/` within your Cygwin installation. You must have write access to the parent directory, and any ACLs on the parent directory will determine access to installed files.

The `Install For` options of `All Users` or `Just Me` should always be left on the default `All Users`, unless you do not have write access to `HKEY_LOCAL_MACHINE` in the registry or the `All Users` Start Menu. This is true even if you are the only

user planning to use Cygwin on the machine. Selecting `Just Me` will cause problems for programs such as `crond` and `sshd`. If you do not have the necessary permissions, but still want to use these programs, consult the Cygwin mailing list archives about others' experiences.

### 2.1.3 Local Package Directory

The `Local Package Directory` is the cache where `setup.exe` stores the packages before they are installed. The cache must not be the same folder as the Cygwin root. Within the cache, a separate directory is created for each Cygwin mirror, which allows `setup.exe` to use multiple mirrors and custom packages. After installing Cygwin, the cache is no longer necessary, but you may want to retain the packages as backups, for installing Cygwin to another system, or in case you need to reinstall a package.

### 2.1.4 Connection Method

The `Direct Connection` method of downloading will directly download the packages, while the `IE5` method will leverage your IE5 cache for performance. If your organisation uses a proxy server or auto-configuration scripts, the `IE5` method also uses these settings. If you have a proxy server, you can manually type it into the `Use Proxy` section. Unfortunately, `setup.exe` does not currently support password authorization for proxy servers.

### 2.1.5 Choosing Mirrors

Since there is no way of knowing from where you will be downloading Cygwin, you need to choose at least one mirror site. Cygwin mirrors are geographically distributed around the world; check the list at <http://cygwin.com/mirrors.html> to find one near you. You can select multiple mirrors by holding down `CTRL` and clicking on each one. If you have the URL of an unlisted mirror (for example, if your organization has an internal Cygwin mirror) you can add it.

### 2.1.6 Choosing Packages

For each selected mirror site, `setup.exe` downloads a small text file called `setup.bz2` that contains a list of packages available from that site along with some basic information about each package which `setup.exe` parses and uses to create the chooser window. For details about the format of this file, see the [setup.exe homepage](#).

The chooser is the most complex part of `setup.exe`. Packages are grouped into categories, and one package may belong to multiple categories (assigned by the volunteer package maintainer). Each package can be found under any of those categories in the hierarchical chooser view. By default, `setup.exe` will install only the packages in the `Base` category and their dependencies, resulting in a minimal Cygwin installation. However, this will not include many commonly used tools such as `gcc` (which you will find in the `Devel` category). Since `setup.exe` automatically selects dependencies, be careful not to unselect any required packages. In particular, everything in the `Base` category is required.

You can change `setup.exe`'s view style, which is helpful if you know the name of a package you want to install but not which category it is in. Click on the `View` button and it will rotate between `Category` (the default), `Full` (all packages), and `Pending` (only packages to be installed, removed or upgraded). If you are familiar with Unix, you will probably want to at least glance through the `Full` listing for your favorite tools.

Once you have an existing Cygwin installation, the `setup.exe` chooser is also used to manage your Cygwin installation. Information on installed packages is kept in the `/etc/setup/` directory of your Cygwin installation; if `setup.exe` cannot find this directory it will act as if you have no Cygwin installation. If `setup.exe` finds a newer version of an installed package available, it will automatically mark it to be upgraded. To `Uninstall`, `Reinstall`, or get the `Source` for an existing package, click on `Keep` to toggle it. Also, to avoid the need to reboot after upgrading, make sure to close all Cygwin windows and stop all Cygwin processes before `setup.exe` begins to install the upgraded package.

To avoid unintentionally upgrading, use the `Pending` view to see which packages have been marked for upgrading. If you don't want to upgrade a package, click on the new version number to toggle it until it says `Keep`. All packages can be set to stay at the installed version by pressing the `Keep` button in the top right part of the chooser window.

A previous version of each package is usually available, in case downgrading is required to avoid a serious bug in the current version of the package. Packages also occasionally have testing (or "experimental") versions available. Previous and experimental versions can be chosen by clicking on the package's `New` column until the required version appears.

All available experimental packages can be selected by pressing the `Exp` in the top right part of the chooser window. Be warned, however, that the next time you run **setup.exe** it will try to replace all old or experimental versions with the current version, unless told otherwise.

### 2.1.7 Download and Installation Progress

First, **setup.exe** will download all selected packages to the local directory chosen earlier. Before installing, **setup.exe** performs a checksum on each package. If the local directory is a slow medium (such as a network drive) this can take a long time. During the download and installation, **setup.exe** shows progress bars for the current task and total remaining disk space.

### 2.1.8 Shortcuts

You may choose to install "Cygwin Terminal" shortcuts on the Desktop and/or Start Menu. These shortcuts run **mintty**, which will start your default shell as specified in `/etc/passwd`.

### 2.1.9 Post-Install Scripts

Last of all, **setup.exe** will run any post-install scripts to finish correctly setting up installed packages. Since each script is run separately, several windows may pop up. If you are interested in what is being done, see the Cygwin Package Contributor's Guide at <http://cygwin.com/setup.html>. When the last post-install script is completed, **setup.exe** will display a box announcing the completion. A few packages, such as the OpenSSH server, require some manual site-specific configuration. Relevant documentation can be found in the `/usr/doc/Cygwin/` or `/usr/share/doc/Cygwin/` directory.

### 2.1.10 Troubleshooting

Unfortunately, the complex setup process means that odd problems can occur. If you're having trouble downloading packages, it may be network congestion, so try a different mirror and/or a different protocol (i.e., HTTP instead of FTP). If you notice something is not working after running setup, you can check the **setup.exe** log file at `/var/log/setup.log.full`. Make a backup of this file before running **setup.exe** again, and follow the steps for [Reporting Problems with Cygwin](#).

## 2.2 Environment Variables

### 2.2.1 Overview

All Windows environment variables are imported when Cygwin starts. Apart from that, you may wish to specify settings of several important environment variables that affect Cygwin's operation.

The `CYGWIN` variable is used to configure a few global settings for the Cygwin runtime system. Typically you can leave `CYGWIN` unset, but if you want to set one or more options, you can set it using a syntax like this, depending on the shell in which you're setting it. Here is an example in CMD syntax:

```
C:\> set CYGWIN=error_start:C:\cygwin\bin\gdb.exe glob
```

This is, of course, just an example. For the recognized settings of the `CYGWIN` environment variable, see [Section 3.5](#).

Locale support is controlled by the `LANG` and `LC_XXX` environment variables. Since Cygwin 1.7.2, all of them are honored and have a meaning. For a more detailed description see [Section 2.4](#).

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:\Windows\system32;C:\Windows`) to UNIX format (e.g., `/cygdrive/c/Windows/system32:/cygdrive/c/Windows`) when a Cygwin process first starts. Set it so that it contains at least the `x:\cygwin\bin` directory where "x:\cygwin" is the "root" of your cygwin installation if you wish to use cygwin tools outside of bash. This is usually done by the batch file you're starting your shell with.



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. It's usually set in the shell profile scripts in the `/etc` directory.

The `TERM` environment variable specifies your terminal type. It is automatically set to `cygwin` if you have not set it to something else.

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.

In addition to `PATH`, `HOME`, and `LD_LIBRARY_PATH`, there are three other environment variables which, if they exist in the Windows environment, are converted to UNIX format: `TMPDIR`, `TMP`, and `TEMP`. The first is not set by default in the Windows environment but the other two are, and they point to the default Windows temporary directory. If set, these variables will be used by some Cygwin applications, possibly with unexpected results. You may therefore want to unset them by adding the following two lines to your `~/ .bashrc` file:

```
unset TMP
unset TEMP
```

This is done in the default `~/ .bashrc` file. Alternatively, you could set `TMP` and `TEMP` to point to `/tmp` or to any other temporary directory of your choice. For example:

```
export TMP=/tmp
export TEMP=/tmp
```

## 2.2.2 Restricted Win32 environment

There is a restriction when calling Win32 API functions which require a fully set up application environment. Cygwin maintains its own environment in POSIX style. The Win32 environment is usually stripped to a bare minimum and not at all kept in sync with the Cygwin POSIX environment.

If you need the full Win32 environment set up in a Cygwin process, you have to call

```
#include <sys/cygwin.h>

cygwin_internal (CW_SYNC_WINENV);
```

to synchronize the Win32 environment with the Cygwin environment. Note that this only synchronizes the Win32 environment once with the Cygwin environment. Later changes using the `setenv` or `putenv` calls are not reflected in the Win32 environment. In these cases, you have to call the aforementioned `cygwin_internal` call again.

## 2.3 Changing Cygwin's Maximum Memory

Cygwin's heap is extensible. However, it does start out at a fixed size and attempts to extend it may run into memory which has been previously allocated by Windows. In some cases, this problem can be solved by changing a field in the file header which is utilized by Cygwin since version 1.7.10 to keep the initial size of the application heap. If the field contains 0, which is the default, the application heap defaults to a size of 384 Megabyte. If the field is set to any other value between 4 and 2048, Cygwin tries to reserve as much Megabytes for the application heap. The field used for this is the "LoaderFlags" field in the NT-specific PE header structure (`(IMAGE_NT_HEADER)->OptionalHeader.LoaderFlags`).

This value can be changed for any executable by using a more recent version of the **peflags** tool from the `rebase` Cygwin package. Example:

```
$ peflags --cygwin-heap foo.exe
foo.exe: initial Cygwin heap size: 0 (0x0) MB
$ peflags --cygwin-heap=500 foo.exe
foo.exe: initial Cygwin heap size: 500 (0x1f4) MB
```

Heap memory can be allocated up to the size of the biggest available free block in the processes virtual memory (VM). By default, the VM per process is 2 GB for 32 processes. To get more VM for a process, the executable must have the "large address aware" flag set in the file header. You can use the aforementioned **peflags** tool to set this flag. On 64 bit systems this results in a 4 GB VM for a process started from that executable. On 32 bit systems you also have to prepare the system to allow up to 3 GB per process. See the Microsoft article [4-Gigabyte Tuning](#) for more information.

#### Note

Older Cygwin releases only supported a global registry setting to change the initial heap size for all Cygwin processes. This setting is not used anymore. However, if you're running an older Cygwin release than 1.7.10, you can add the `DWORD` value `heap_chunk_in_mb` and set it to the desired memory limit in decimal MB. You have to stop all Cygwin processes for this setting to have any effect. It is preferred to do this in Cygwin using the **regtool** program included in the Cygwin package. (see [?]) This example sets the memory limit to 1024 MB for all Cygwin processes (use `HKCU` instead of `HKLM` if you want to set this only for the current user):

```
$ regtool -i set /HKLM/Software/Cygwin/heap_chunk_in_mb 1024
$ regtool -v list /HKLM/Software/Cygwin
```

## 2.4 Internationalization

### 2.4.1 Overview

Internationalization support is controlled by the `LANG` and `LC_XXX` environment variables. You can set all of them but Cygwin itself only honors the variables `LC_ALL`, `LC_CTYPE`, and `LANG`, in this order, according to the POSIX standard. The content of these variables should follow the POSIX standard for a locale specifier. The correct form of a locale specifier is

```
language[_TERRITORY][.charset][@modifier]
```

"language" is a lowercase two character string per ISO 639-1, or, if there is no ISO 639-1 code for the language (for instance, "Lower Sorbian"), a three character string per ISO 639-3.

"TERRITORY" is an uppercase two character string per ISO 3166, charset is one of a list of supported character sets. The modifier doesn't matter here (though some are recognized, see below). If you're interested in the exact description, you can find it in the online publication of the POSIX manual pages on the homepage of the [Open Group](#).

Typical locale specifiers are

```
"de_CH"      language = German, territory = Switzerland, default charset
"fr_FR.UTF-8" language = french, territory = France, charset = UTF-8
"ko_KR.eucKR" language = korean, territory = South Korea, charset = eucKR
"syr_SY"      language = Syriac, territory = Syria, default charset
```

If the locale specifier does not follow the above form, Cygwin checks if the locale is one of the locale aliases defined in the file `/usr/share/locale/locale.alias`. If so, and if the replacement localename is supported by the underlying Windows, the locale is accepted, too. So, given the default content of the `/usr/share/locale/locale.alias` file, the below examples would be valid locale specifiers as well.

```
"catalan"      defined as "ca_ES.ISO-8859-1" in locale.alias
"japanese"     defined as "ja_JP.eucJP"      in locale.alias
"turkish"      defined as "tr_TR.ISO-8859-9" in locale.alias
```

The file `/usr/share/locale/locale.alias` is provided by the `gettext` package under Cygwin.

At application startup, the application's locale is set to the default "C" or "POSIX" locale. Under Cygwin 1.7.2 and later, this locale defaults to the ASCII character set on the application level. If you want to stick to the "C" locale and only change to another charset, you can define this by setting one of the locale environment variables to "C.charset". For instance

```
"C.ISO-8859-1"
```

---

**Note**

The default locale in the absence of the aforementioned locale environment variables is "C.UTF-8".

---

Windows uses the UTF-16 charset exclusively to store the names of any object used by the Operating System. This is especially important with filenames. Cygwin uses the setting of the locale environment variables `LC_ALL`, `LC_CTYPE`, and `LANG`, to determine how to convert Windows filenames from their UTF-16 representation to the singlebyte or multibyte character set used by Cygwin.

The setting of the locale environment variables at process startup is effective for Cygwin's internal conversions to and from the Windows UTF-16 object names for the entire lifetime of the current process. Changing the environment variables to another value changes the way filenames are converted in subsequently started child processes, but not within the same process.

However, even if one of the locale environment variables is set to some other value than "C", this does *only* affect how Cygwin itself converts filenames. As the POSIX standard requires, it's the application's responsibility to activate that locale for its own purposes, typically by using the call

```
setlocale (LC_ALL, "");
```

early in the application code. Again, so that this doesn't get lost: If the application calls `setlocale` as above, and there is none of the important locale variables set in the environment, the locale is set to the default locale, which is "C.UTF-8".

But what about applications which are not locale-aware? Per POSIX, they are running in the "C" or "POSIX" locale, which implies the ASCII charset. The Cygwin DLL itself, however, will nevertheless use the locale set in the environment (or the "C.UTF-8" default locale) for converting filenames etc.

When the locale in the environment specifies an ASCII charset, for example "C" or "en\_US.ASCII", Cygwin will still use UTF-8 under the hood to translate filenames. This allows for easier interoperability with applications running in the default "C.UTF-8" locale.

Starting with Cygwin 1.7.2, the language and territory are used to fetch locale-dependent information from Windows. If the language and territory are not known to Windows, the `setlocale` function fails.

The following modifiers are recognized. Any other modifier is simply ignored for now.

- For locales which use the Euro (EUR) as currency, the modifier "@euro" can be added to enforce usage of the ISO-8859-15 character set, which includes a character for the "Euro" currency sign.
- The default script used for all Serbian language locales (`sr_BA`, `sr_ME`, `sr_RS`, and the deprecated `sr_CS` and `sr_SP`) is cyrillic. With the "@latin" modifier it gets switched to the latin script with the respective collation behaviour.
- The default charset of the "be\_BY" locale (Belarusian/Belarus) is CP1251. With the "@latin" modifier it's UTF-8.
- The default charset of the "tt\_RU" locale (Tatar/Russia) is ISO-8859-5. With the "@iqtelif" modifier it's UTF-8.
- The default charset of the "uz\_UZ" locale (Uzbek/Uzbekistan) is ISO-8859-1. With the "@cyrillic" modifier it's UTF-8.
- There's a class of characters in the Unicode character set, called the "CJK Ambiguous Width" characters. For these characters, the width returned by the `wcwidth/wcswidth` functions is usually 1. This can be a problem with East-Asian languages, which historically use character sets where these characters have a width of 2. Therefore, `wcwidth/wcswidth` return 2 as the width of these characters when an East-Asian charset such as GBK or SJIS is selected, or when UTF-8 is selected and the language is specified as "zh" (Chinese), "ja" (Japanese), or "ko" (Korean). This is not correct in all circumstances, hence the locale modifier "@cjknnarrow" can be used to force `wcwidth/wcswidth` to return 1 for the ambiguous width characters.

## 2.4.2 How to set the locale

- Assume that you've set one of the aforementioned environment variables to some valid POSIX locale value, other than "C" and "POSIX". Assume further that you're living in Japan. You might want to use the language code "ja" and the territory "JP", thus setting, say, `LANG` to "ja\_JP". You didn't set a character set, so what will Cygwin use now? Starting with Cygwin 1.7.2, the default character set is determined by the default Windows ANSI codepage for this language and territory. Cygwin uses a character set which is the typical Unix-equivalent to the Windows ANSI codepage. For instance:

```
"en_US"    ISO-8859-1
"el_GR"    ISO-8859-7
"pl_PL"    ISO-8859-2
"pl_PL@euro" ISO-8859-15
"ja_JP"    EUCJP
"ko_KR"    EUCKR
"te_IN"    UTF-8
```

- You don't want to use the default character set? In that case you have to specify the charset explicitly. For instance, assume you're from Japan and don't want to use the japanese default charset EUC-JP, but the Windows default charset SJIS. What you can do, for instance, is to set the `LANG` variable in the **mintty** Cygwin Terminal in the "Text" section of its "Options" dialog. If you're starting your Cygwin session via a batch file or a shortcut to a batch file, you can also just set `LANG` there:

```
@echo off

C:
chdir C:\cygwin\bin
set LANG=ja_JP.SJIS
bash --login -i
```

---

**Note**

For a list of locales supported by your Windows machine, use the new **locale -a** command, which is part of the Cygwin package. For a description see [?]

---

---

**Note**

For a list of supported character sets, see Section [2.4.5](#)

---

- Last, but not least, most singlebyte or doublebyte charsets have a big disadvantage. Windows filesystems use the Unicode character set in the UTF-16 encoding to store filename information. Not all characters from the Unicode character set are available in a singlebyte or doublebyte charset. While Cygwin has a workaround to access files with unusual characters (see Section [3.4.4](#)), a better workaround is to use always the UTF-8 character set.

*UTF-8 is the only multibyte character set which can represent every Unicode character.*

```
set LANG=es_MX.UTF-8
```

For a description of the Unicode standard, see the homepage of the [Unicode Consortium](#).

### 2.4.3 The Windows Console character set

Sometimes the Windows console is used to run Cygwin applications. While terminal emulations like the Cygwin Terminal **mintty** or **xterm** have a distinct way to set the character set used for in- and output, the Windows console hasn't such a way, since it's not an application in its own right.

This problem is solved in Cygwin as follows. When a Cygwin process is started in a Windows console (either explicitly from `cmd.exe`, or implicitly by, for instance, running the `C:\cygwin\Cygwin.bat` batch file), the Console character set is determined by the setting of the aforementioned internationalization environment variables, the same way as described in Section [2.4.2](#).

What is that good for? Why not switch the console character set with the applications requirements? After all, the application knows if it uses localization or not. However, what if a non-localized application calls a remote application which itself is localized? This can happen with **ssh** or **rlogin**. Both commands don't have and don't need localization and they never call `setlocale`. Setting one of the internationalization environment variable to the same charset as the remote machine before starting **ssh** or **rlogin** fixes that problem.

---

## 2.4.4 Potential Problems when using Locales

You can set the above internationalization variables not only when starting the first Cygwin process, but also in your Cygwin shell on the fly, even switch to yet another character set, and yet another. In bash for instance:

```
bash$ export LC_CTYPE="nl_BE.UTF-8"
```

However, here's a problem. At the start of the first Cygwin process in a session, the Windows environment is converted from UTF-16 to UTF-8. The environment is another of the system objects stored in UTF-16 in Windows.

As long as the environment only contains ASCII characters, this is no problem at all. But if it contains native characters, and you're planning to use, say, GBK, the environment will result in invalid characters in the GBK charset. This would be especially a problem in variables like `PATH`. To circumvent the worst problems, Cygwin converts the `PATH` environment variable to the charset set in the environment, if it's different from the UTF-8 charset.

### Note

Per POSIX, the name of an environment variable should only consist of valid ASCII characters, and only of uppercase letters, digits, and the underscore for maximum portability.

Symbolic links, too, may pose a problem when switching charsets on the fly. A symbolic link contains the filename of the target file the symlink points to. When a symlink had been created with older versions of Cygwin, the current ANSI or OEM character set had been used to store the target filename, dependent on the old `CYGWIN` environment variable setting `codepage` (see Section 3.5.2. If the target filename contains non-ASCII characters and you use another character set than your default ANSI/OEM charset, the target filename of the symlink is now potentially an invalid character sequence in the new character set. This behaviour is not different from the behaviour in other Operating Systems. So, if you suddenly can't access a symlink anymore which worked all these years before, maybe it's because you switched to another character set. This doesn't occur with symlinks created with Cygwin 1.7 or later.

Another problem you might encounter is that older versions of Windows did not install all charsets by default. If you are running Windows XP or older, you can open the "Regional and Language Options" portion of the Control Panel, select the "Advanced" tab, and select entries from the "Code page conversion tables" list. The following entries are useful to cygwin: 932/SJIS, 936/GBK, 949/EUC-KR, 950/Big5, 20932/EUC-JP.

## 2.4.5 List of supported character sets

Last but not least, here's the list of currently supported character sets. The left-hand expression is the name of the charset, as you would use it in the internationalization environment variables as outlined above. Note that charset specifiers are case-insensitive. `EUCJP` is equivalent to `eucJP` or `eUcJp`. Writing the charset in the exact case as given in the list below is a good convention, though.

The right-hand side is the number of the equivalent Windows codepage as well as the Windows name of the codepage. They are only noted here for reference. Don't try to use the bare codepage number or the Windows name of the codepage as charset in locale specifiers, unless they happen to be identical with the left-hand side. Especially in case of the "CPxxx" style charsets, always use them with the trailing "CP".

This works:

```
set LC_ALL=en_US.CP437
```

This does *not* work:

```
set LC_ALL=en_US.437
```

You can find a full list of Windows codepages on the Microsoft MSDN page [Code Page Identifiers](#).

Charset	Codepage
-----	-----
ASCII	20127 (US_ASCII)

CP437	437 (OEM United States)
CP720	720 (DOS Arabic)
CP737	737 (OEM Greek)
CP775	775 (OEM Baltic)
CP850	850 (OEM Latin 1, Western European)
CP852	852 (OEM Latin 2, Central European)
CP855	855 (OEM Cyrillic)
CP857	857 (OEM Turkish)
CP858	858 (OEM Latin 1 + Euro Symbol)
CP862	862 (OEM Hebrew)
CP866	866 (OEM Russian)
CP874	874 (ANSI/OEM Thai)
CP932	932 (Shift_JIS, not exactly identical to SJIS)
CP1125	1125 (OEM Ukraine)
CP1250	1250 (ANSI Central European)
CP1251	1251 (ANSI Cyrillic)
CP1252	1252 (ANSI Latin 1, Western European)
CP1253	1253 (ANSI Greek)
CP1254	1254 (ANSI Turkish)
CP1255	1255 (ANSI Hebrew)
CP1256	1256 (ANSI Arabic)
CP1257	1257 (ANSI Baltic)
CP1258	1258 (ANSI/OEM Vietnamese)
ISO-8859-1	28591 (ISO-8859-1)
ISO-8859-2	28592 (ISO-8859-2)
ISO-8859-3	28593 (ISO-8859-3)
ISO-8859-4	28594 (ISO-8859-4)
ISO-8859-5	28595 (ISO-8859-5)
ISO-8859-6	28596 (ISO-8859-6)
ISO-8859-7	28597 (ISO-8859-7)
ISO-8859-8	28598 (ISO-8859-8)
ISO-8859-9	28599 (ISO-8859-9)
ISO-8859-10	- (not available)
ISO-8859-11	- (not available)
ISO-8859-13	28603 (ISO-8859-13)
ISO-8859-14	- (not available)
ISO-8859-15	28605 (ISO-8859-15)
ISO-8859-16	- (not available)
Big5	950 (ANSI/OEM Traditional Chinese)
EUC-CN or euc-CN	936 (ANSI/OEM Simplified Chinese)
EUCJP or euc-JP	20932 (EUC Japanese)
EUCKR or euc-KR	949 (EUC Korean)
GB2312	936 (ANSI/OEM Simplified Chinese)
GBK	936 (ANSI/OEM Simplified Chinese)
GEORGIAN-PS	- (not available)
KOI8-R	20866 (KOI8-R Russian Cyrillic)
KOI8-U	21866 (KOI8-U Ukrainian Cyrillic)
PT154	- (not available)
SJIS	- (not available, almost, but not exactly CP932)
TIS620 or TIS-620	874 (ANSI/OEM Thai)
UTF-8 or utf8	65001 (UTF-8)

## 2.5 Customizing bash

To set up bash so that cut and paste work properly, click on the "Properties" button of the window, then on the "Misc" tab. Make sure that "QuickEdit mode" and "Insert mode" are checked. 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, but requires that you set HOME in the Windows environment.

Your home directory should contain three initialization files that control the behavior of bash. They are `.profile`, `.bashrc` and `.inputrc`. The Cygwin base installation creates stub files when you start bash for the first time.

`.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**. 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. For full details see the Function and Variable Index section of the GNU readline manual. Consider the following settings:

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

The first command makes filename completion case insensitive, which can be convenient in a Windows environment. The next three commands allow **bash** to display 8-bit characters, useful for languages with accented characters. Note that tools that do not use readline for display, such as **less** and **ls**, require additional settings, which could be put in your `.bashrc`:

```
alias less='/bin/less -r'
alias ls='/bin/ls -F --color=tty --show-control-chars'
```

## 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 POSIX- and Win32-style paths. Directory delimiters may be either forward slashes or backslashes. Paths using backslashes or starting with a drive letter are always handled as Win32 paths. POSIX paths must only use forward slashes as delimiter, otherwise they are treated as Win32 paths and file access might fail in surprising ways.

---

**Note**

The usage of Win32 paths, though possible, is deprecated, since it circumvents important internal path handling mechanisms. See Section 3.1.5 and Section 3.1.6 for more information.

---

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 existence 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 from POSIX to Win32 paths as necessary.

#### 3.1.2 The Cygwin Mount Table

The `/etc/fstab` file is used to map Win32 drives and network shares into Cygwin's internal POSIX directory tree. This is a similar concept to the typical UNIX `fstab` file. The mount points stored in `/etc/fstab` are globally set for all users. Sometimes there's a requirement to have user specific mount points. The Cygwin DLL supports user specific `fstab` files. These are stored in the directory `/etc/fstab.d` and the name of the file is the Cygwin username of the user, as it's stored in the `/etc/passwd` file. The structure of the user specific file is identical to the system-wide `fstab` file.

The file `fstab` contains descriptive information about the various file systems. `fstab` is only read by programs, and not written; it is the duty of the system administrator to properly create and maintain this file. Each filesystem is described on a separate line; fields on each line are separated by tabs or spaces. Lines starting with `'#'` are comments.

The first field describes the block special device or remote filesystem to be mounted. On Cygwin, this is the native Windows path which the mount point links in. As path separator you **MUST** use a slash. Usage of a backslash might lead to unexpected results. UNC paths (using slashes, not backslashes) are allowed. If the path contains spaces these can be escaped as `'\040'`.

---



The second field describes the mount point for the filesystem. If the name of the mount point contains spaces these can be escaped as `'\040'`.

The third field describes the type of the filesystem. Cygwin supports any string here, since the file system type is usually not evaluated. So it doesn't matter if you write `FAT` into this field even if the filesystem is `NTFS`. Cygwin figures out the filesystem type and its capabilities by itself.

The only exception is the file system type `cygdrive`. This type is used to set the `cygdrive` prefix. For a description of the `cygdrive` prefix see Section 3.1.4

The fourth field describes the mount options associated with the filesystem. It is formatted as a comma separated list of options. It contains at least the type of mount (binary or text) plus any additional options appropriate to the filesystem type. Recognized options are `binary`, `text`, `nouser`, `user`, `exec`, `notexec`, `cygexec`, `nosuid`, `posix=[01]`. The meaning of the options is as follows.

```

acl      - Cygwin uses the filesystem's access control lists (ACLs) to
           implement real POSIX permissions (default). This flag only
           affects filesystems supporting ACLs (NTFS, for instance) and
           is ignored otherwise.
auto     - Ignored.
binary   - Files default to binary mode (default).
bind     - Allows to remount part of the file hierarchy somewhere else.
           In contrast to other entries, the first field in the fstab
           line specifies an absolute POSIX path. This path is remounted
           to the POSIX path specified as the second path. The conversion
           to a Win32 path is done on the fly. Only the root path and
           paths preceding the bind entry in the fstab file are used to
           convert the POSIX path in the first field to an absolute Win32
           path. Note that symlinks are ignored while performing this path
           conversion.
cygexec  - Treat all files below mount point as cygwin executables.
dos      - Always convert leading spaces and trailing dots and spaces to
           characters in the UNICODE private use area. This allows to use
           broken filesystems which only allow DOS filenames, even if they
           are not recognized as such by Cygwin.
exec     - Treat all files below mount point as executable.
ihash    - Always fake inode numbers rather than using the ones returned
           by the filesystem. This allows to use broken filesystems which
           don't return unambiguous inode numbers, even if they are not
           recognized as such by Cygwin.
noacl    - Cygwin ignores filesystem ACLs and only fakes a subset of
           permission bits based on the DOS readonly attribute. This
           behaviour is the default on FAT and FAT32. The flag is
           ignored on NFS filesystems.
nosuid   - No suid files are allowed (currently unimplemented).
notexec  - Treat all files below mount point as not executable.
nouser   - Mount is a system-wide mount.
override - Force the override of an immutable mount point (currently "/").
posix=0  - Switch off case sensitivity for paths under this mount point
           (default for the cygdrive prefix).
posix=1  - Switch on case sensitivity for paths under this mount point
           (default for all other mount points).
sparse   - Switch on support for sparse files. This option only makes
           sense on NTFS and then only if you really need sparse files.
           Cygwin does not try to create sparse files by default for
           performance reasons.
text     - Files default to CRLF text mode line endings.
user     - Mount is a user mount.
```

While normally the execute permission bits are used to evaluate executability, this is not possible on filesystems which don't support permissions at all (like `FAT`/`FAT32`), or if ACLs are ignored on filesystems supporting them (see the aforementioned `acl` mount option). In these cases, the following heuristic is used to evaluate if a file is executable: Files ending in certain extensions (`.exe`, `.com`, `.bat`, `.btm`, `.cmd`) are assumed to be executable. Files whose first two characters begin with `'#!'` are

also considered to be executable. The `exec` option is used to instruct Cygwin that the mounted file is "executable". If the `exec` option is used with a directory then all files in the directory are executable. This option allows other files to be marked as executable and avoids the overhead of opening each file to check for a `'#!'`. The `cygexec` option is very similar to `exec`, but also prevents Cygwin from setting up commands and environment variables for a normal Windows program, adding another small performance gain. The opposite of these options is the `notexec` option, which means that no files should be marked as executable under that mount point.

A correct root directory is quite essential to the operation of Cygwin. A default root directory is evaluated at startup so a `fstab` entry for the root directory is not necessary. If it's wrong, nothing will work as expected. Therefore, the root directory evaluated by Cygwin itself is treated as an immutable mount point and can't be overridden in `/etc/fstab`... unless you think you really know what you're doing. In this case, use the `override` flag in the options field in the `/etc/fstab` file. Since this is a dangerous thing to do, do so at your own risk.

`/usr/bin` and `/usr/lib` are by default also automatic mount points generated by the Cygwin DLL similar to the way the root directory is evaluated. `/usr/bin` points to the directory the Cygwin DLL is installed in, `/usr/lib` is supposed to point to the `/lib` directory. This choice is safe and usually shouldn't be changed. An `fstab` entry for them is not required.

`nouser` mount points are not overridable by a later call to **mount**. Mount points given in `/etc/fstab` are by default `nouser` mount points, unless you specify the option `user`. This allows the administrator to set certain paths so that they are not overridable by users. In contrast, all mount points in the user specific `fstab` file are `user` mount points.

The fifth and sixth field are ignored. They are so far only specified to keep a Linux-like `fstab` file layout.

Note that you don't have to specify an `fstab` entry for the root dir, unless you want to have the root dir pointing to somewhere entirely different (hopefully you know what you're doing), or if you want to mount the root dir with special options (for instance, as text mount).

Example entries:

- Just a normal mount point:

```
c:/foo /bar fat32 binary 0 0
```

- A mount point for a textmode mount with case sensitivity switched off:

```
C:/foo /bar/baz ntfs text,posix=0 0 0
```

- A mount point for a Windows directory with spaces in it:

```
C:/Documents\040and\040Settings /docs ext3 binary 0 0
```

- A mount point for a remote directory, don't store POSIX permissions in ACLs:

```
//server/share/subdir /srv/subdir smbfs binary,noacl 0 0
```

- This is just a comment:

```
# This is just a comment
```

- Set the `cygdrive` prefix to `/mnt`:

```
none /mnt cygdrive binary 0 0
```

- Remount `/var` to `/usr/var`:

```
/var /usr/var none bind
```

Assuming `/var` points to `C:/cygwin/var`, `/usr/var` now also points to `C:/cygwin/var`. This is equivalent to the Linux `bind` option available since Linux 2.4.0.

Whenever Cygwin generates a Win32 path from a POSIX 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`. This translation is normally only used when trying to derive the POSIX equivalent current directory. Otherwise, the handling of MS-DOS filenames bypasses the mount table.

If you want to see the current set of mount points valid in your session, you can invoke the Cygwin tool **mount** without arguments:

---

**Example 3.1** Displaying the current set of mount points

---

```
bash$ mount
f:/cygwin/bin on /usr/bin type ntfs (binary,auto)
f:/cygwin/lib on /usr/lib type ntfs (binary,auto)
f:/cygwin on / type ntfs (binary,auto)
e:/src on /usr/src type vfat (binary)
c: on /cygdrive/c type ntfs (binary,posix=0,user,noumount,auto)
e: on /cygdrive/e type vfat (binary,posix=0,user,noumount,auto)
```

---

You can also use the **mount** command to add new mount points, and the **umount** to delete them. However, since they are only stored in memory, these mount points will disappear as soon as your last Cygwin process ends. See [?] and [?] for more information.

---

**Note**

When you upgrade an existing older Cygwin installation to Cygwin 1.7, your old system mount points (stored in the `HKEY_LOCAL_MACHINE` branch of your registry) are read by a script and the `/etc/fstab` file is generated from these entries. Note that entries for `/`, `/usr/bin`, and `/usr/lib` are **never** generated.

The old user mount points in your `HKEY_CURRENT_USER` branch of the registry are not used to generate `/etc/fstab`. If you want to create a user specific `/etc/fstab.d/${USER}` file from your old entries, there's a script available which does exactly that for you, `/bin/copy-user-registry-fstab`. Just start the script and it will create your user specific `fstab` file. Stop all your Cygwin processes and restart them, and you can simply use your old user mount points as before.

---

### 3.1.3 UNC paths

Apart from the unified POSIX tree starting at the `/` directory, UNC pathnames starting with two slashes and a server name (`//machine/share/...`) are supported as well. They are handled as POSIX paths if only containing forward slashes. There's also a virtual directory `//` which allows to enumerate the filesystems known to the local machine with `ls`. Same goes for the UNC paths of the type `//machine`, which allow to enumerate the shares provided by the server `machine`. For often used UNC paths it makes sense to add them to the mount table (see Section 3.1.2 so they are included in the unified POSIX path tree.

### 3.1.4 The cygdrive path prefix

As already outlined in Section 1.6.3, you can access arbitrary drives on your system by using the `cygdrive` path prefix. The default value for this prefix is `/cygdrive`, and a path to any drive can be constructed by using the `cygdrive` prefix and appending the drive letter as subdirectory, like this:

```
bash$ ls -l /cygdrive/f/somedir
```

This lists the content of the directory `F:\somedir`.

The `cygdrive` prefix is a virtual directory under which all drives on a system are subsumed. The mount options of the `cygdrive` prefix is used for all file access through the `cygdrive` prefixed drives. For instance, assuming the `cygdrive` mount options are `binary,posix=0`, then any file `/cygdrive/x/file` will be opened in binary mode by default (mount option `binary`), and the case of the filename doesn't matter (mount option `posix=0`).

The `cygdrive` prefix flags are also used for all UNC paths starting with two slashes, unless they are accessed through a mount point. For instance, consider these `/etc/fstab` entries:

---

```
//server/share /mysrv ntfs posix=1,acl 0 0
none /cygdrive cygdrive posix=0,noacl 0 0
```

Assume there's a file `\\server\share\foo` on the share. When accessing it as `/mysrv/foo`, then the flags `posix=1,-acl` of the `/mysrv` mount point are used. When accessing it as `//server/share/foo`, then the flags for the `cygdrive` prefix, `posix=0,noacl` are used.

---

**Note**

This only applies to UNC paths using forward slashes. When using backslashes the flags for native paths are used. See Section 3.1.5.

---

The `cygdrive` prefix may be changed in the `fstab` file as outlined above. Please note that you must not use the `cygdrive` prefix for any other mount point. For instance this:

```
none /cygdrive cygdrive binary 0 0
D: /cygdrive/d somefs text 0 0
```

will not make file access using the `/mnt/d` path prefix suddenly using textmode. If you want to mount any drive explicitly in another mode than the `cygdrive` prefix, use a distinct path prefix:

```
none /cygdrive cygdrive binary 0 0
D: /mnt/d somefs text 0 0
```

### 3.1.5 Using native Win32 paths

Using native Win32 paths in Cygwin, while possible, is generally inadvisable. Those paths circumvent all internal integrity checking and bypass the information given in the Cygwin mount table.

The following paths are treated as native Win32 paths in Cygwin:

- All paths starting with a drive specifier

```
C:\foo
C:/foo
```

- All paths containing at least one backslash as path component

```
C:/foo/bar\baz/...
```

- UNC paths using backslashes

```
\\server\share\...
```

When accessing files using native Win32 paths as above, Cygwin uses a default setting for the mount flags. All paths using DOS notation will be treated as case insensitive, and permissions are just faked as if the underlying drive is a FAT drive. This also applies to NTFS and other filesystems which usually are capable of case sensitivity and storing permissions.

### 3.1.6 Using the Win32 file API in Cygwin applications

Special care must be taken if your application uses Win32 file API functions like `CreateFile` to access files using relative pathnames, or if your application uses functions like `CreateProcess` or `ShellExecute` to start other applications.

When a Cygwin application is started, the Windows idea of the current working directory (CWD) is not necessarily the same as the Cygwin CWD. There are a couple of restrictions in the Win32 API, which disallow certain directories as Win32 CWD:

---

- The Windows subsystem only supports CWD paths of up to 258 chars. This restriction doesn't apply for Cygwin processes, at least not as long as they use the POSIX API (`chdir`, `getcwd`). This means, if a Cygwin process has a CWD using an absolute path longer than 258 characters, the Cygwin CWD and the Windows CWD differ.
- The Win32 API call to set the current directory, `SetCurrentDirectory`, fails for directories for which the user has no permissions, even if the user is an administrator. This restriction doesn't apply for Cygwin processes, if they are running under an administrator account.
- `SetCurrentDirectory` does not support case-sensitive filenames.
- Last, but not least, `SetCurrentDirectory` can't work on virtual Cygwin paths like `/proc` or `/cygdrive`. These paths only exist in the Cygwin realm so they have no meaning to a native Win32 process.

As long as the Cygwin CWD is usable as Windows CWD, the Cygwin and Windows CWDs are in sync within a process. However, if the Cygwin process changes its working directory into one of the directories which are unusable as Windows CWD, we're in trouble. If the process uses the Win32 API to access a file using a relative pathname, the resulting absolute path would not match the expectations of the process. In the worst case, the wrong files are deleted.

To work around this problem, Cygwin sets the Windows CWD to a special directory in this case. This special directory points to a virtual filesystem within the native NT namespace (`\\??\PIPE\`). Since it's not a real filesystem, the deliberate effect is that a call to, for instance, `CreateFile ("foo", ...)` will fail, as long as the process's CWD doesn't work as Windows CWD.

So, in general, don't use the Win32 file API in Cygwin applications. If you **really** need to access files using the Win32 API, or if you **really** have to use `CreateProcess` to start applications, rather than the POSIX `exec(3)` family of functions, you have to make sure that the Cygwin CWD is set to some directory which is valid as Win32 CWD.

### 3.1.7 Additional Path-related Information

The **cygpath** program provides the ability to translate between Win32 and POSIX pathnames in shell scripts. See [?] 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:/cygwin/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 on writing in text mode, a NL (`\n`, `^J`) is transformed into the sequence CR (`\r`, `^M`) NL.

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 **sed**, **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 filename is specified as a POSIX path and it 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 is specified via a MS-DOS pathname (i.e., it contains a backslash or a colon), the default is binary.
- c. Pipes, sockets and non-file devices are opened in binary mode. For pipes opened through the `pipe()` system call you can use the `setmode()` function (see Section 3.2.4) to switch to textmode. For pipes opened through `popen()`, you can simply specify text or binary mode just like in calls to `fopen()`.
- d. Sockets and other non-file devices are always opened in binary mode.
- e. When redirecting, the Cygwin shells uses rules (a-d). 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.

The programs **u2d** and **d2u** can be used to add or remove CR's from a file. **u2d** add's CR's before a NL. **d2u** removes CR's. Use the `--help` option to these commands for more information.

### 3.2.3 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. Most programs included in the official Cygwin distributions should work well in the default mode.

Binmode is the best choice usually since it's faster and easier to handle, unless you want to exchange files with native Win32 applications. It makes most sense to keep the Cygwin distribution and your Cygwin home directory in binmode and generate text files in binmode (with UNIX LF lineendings). Most Windows applications can handle binmode files just fine. A notable exception is the mini-editor **Notepad**, which handles UNIX lineendings incorrectly and only produces output files with DOS CRLF lineendings.

You can convert files between CRLF and LF lineendings by using certain tools in the Cygwin distribution like **d2u** and **u2d** from the `cygutils` package. You can also specify a directory in the mount table to be mounted in textmode so you can use that directory for exchange purposes.

As application programmer you can decide on a file by file base, or you can specify default open modes depending on the purpose for which the application open files. See the next section for a description of your choices.

### 3.2.4 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`.

The `mkstemp()` and `mkstemp()` calls force binary mode. Use `mkostemp()` or `mkostemp()` with the same flags as `open()` for more control on temporary files.

In the `fopen()` and `popen()` function calls, binary mode can be specified by adding a `b` to the mode string. Text mode is specified by adding a `t` to the mode string.

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.

There's also a convenient way to set the default open modes used in an application by just linking against various object files provided by Cygwin. For instance, if you want to make sure that all files are always opened in binary mode by an application, regardless of the mode of the underlying mount point, just add the file `/lib/binmode.o` to the link stage of the application in your project, like this:

```
$ gcc my_tiny_app.c /lib/binmode.o -o my_tiny_app
```

Starting with Cygwin 1.7.7, you can use the even simpler:

```
$ gcc my_tiny_app.c -lbinmode -o my_tiny_app
```

This adds code which sets the default open mode for all files opened by **my\_tiny\_app** to binary for reading and writing.

Cygwin provides the following libraries and object files to set the default open mode just by linking an application against them:

- /lib/libautomode.a      - Open files for reading in textmode,  
/lib/automode.o        open files for writing in binary mode
- /lib/libbinmode.a      - Open files for reading and writing in binary mode  
/lib/binmode.o
- /lib/libtextmode.a     - Open files for reading and writing in textmode  
/lib/textmode.o
- /lib/libtextreadmode.a - Open files for reading in textmode,  
/lib/textreadmode.o    keep default behaviour for writing.

### 3.3 File permissions

On FAT or FAT32 filesystems, files are always readable, and Cygwin uses the DOS read-only attribute 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.

On NTFS, file permissions are evaluated using the Access Control Lists (ACLs) attached to a file. This can be switched off by using the "noacl" option to the respective mount point in the `/etc/fstab` or `/etc/fstab.d/$USER` file. For more information on file permissions, see Section 3.6.

On NFS shares, file permissions are exactly the POSIX permissions transmitted from the server using the NFSv3 protocol, if the NFS client is the one from Microsoft's "Services For Unix", or the one built into Windows Vista or later.

Only the user and group ownership is not necessarily correct.

### 3.4 Special filenames

#### 3.4.1 Special files in /etc

Certain files in Cygwin's `/etc` directory are read by Cygwin before the mount table has been established. The list of files is

```
/etc/fstab  
/etc/fstab.d/$USER  
/etc/passwd  
/etc/group
```

These files are read using native Windows NT functions which have no notion of Cygwin symlinks or POSIX paths. For that reason there are a few requirements as far as `/etc` is concerned.

To access these files, the Cygwin DLL evaluates its own full Windows path, strips off the innermost directory component and adds "etc". Let's assume the Cygwin DLL is installed as `C:\cygwin\bin\cygwin1.dll`. First the DLL name as

well as the innermost directory (`bin`) is stripped off: `C:\cygwin\`. Then `"etc"` and the filename to look for is attached: `C:\cygwin\etc\fstab`. So the `/etc` directory must be parallel to the directory in which the `cygwin1.dll` exists and `/etc` must not be a Cygwin symlink pointing to another directory. Consequentially none of the files from the above list, including the directory `/etc/fstab.d` is allowed to be a Cygwin symlink either.

However, native NTFS symlinks and reparse points are transparent when accessing the above files so all these files as well as `/etc` itself may be NTFS symlinks or reparse points.

Last but not least, make sure that these files are world-readable. Every process of any user account has to read these files potentially, so world-readability is essential. The only exception are the user specific files `/etc/fstab.d/$USER`, which only have to be readable by the `$USER` user account itself.

### 3.4.2 Invalid filenames

Filenames invalid under Win32 are not necessarily invalid under Cygwin since release 1.7.0. There are a few rules which apply to Windows filenames. Most notably, DOS device names like `AUX`, `COM1`, `LPT1` or `PRN` (to name a few) cannot be used as filename or extension in a native Win32 application. So filenames like `prn.txt` or `foo.aux` are invalid filenames for native Win32 applications.

This restriction doesn't apply to Cygwin applications. Cygwin can create and access files with such names just fine. Just don't try to use these files with native Win32 applications.

### 3.4.3 Forbidden characters in filenames

Some characters are disallowed in filenames on Windows filesystems. These forbidden characters are the ASCII control characters from ASCII value 1 to 31, plus the following characters which have a special meaning in the Win32 API:

```
" * : < > ? | \
```

Cygwin can't fix this, but it has a method to workaround this restriction. All of the above characters, except for the backslash, are converted to special UNICODE characters in the range `0xf000` to `0xf0ff` (the "Private use area") when creating or accessing files.

The backslash has to be exempt from this conversion, because Cygwin accepts Win32 filenames including backslashes as path separators on input. Converting backslashes using the above method would make this impossible.

Additionally Win32 filenames can't contain trailing dots and spaces for DOS backward compatibility. When trying to create files with trailing dots or spaces, all of them are removed before the file is created. This restriction only affects native Win32 applications. Cygwin applications can create and access files with trailing dots and spaces without problems.

An exception from this rule are some network filesystems (NetApp, NWFS) which choke on these filenames. They return with an error like "No such file or directory" when trying to create such files. Starting with Cygwin 1.7.6, Cygwin recognizes these filesystems and works around this problem by applying the same rule as for the other forbidden characters. Leading spaces and trailing dots and spaces will be converted to UNICODE characters in the private use area. This behaviour can be switched on explicitly for a filesystem or a directory tree by using the mount option `dos`.

### 3.4.4 Filenames with unusual (foreign) characters

Windows filesystems use Unicode encoded as UTF-16 to store filename information. If you don't use the UTF-8 character set (see Section 2.4) then there's a chance that a filename is using one or more characters which have no representation in the character set you're using.

---

#### Note

In the default "C" locale, Cygwin creates filenames using the UTF-8 charset. This will always result in some valid filename by default, but again might impose problems when switching to a non-"C" or non-"UTF-8" charset.

---



---

**Note**

To avoid this scenario altogether, always use UTF-8 as the character set.

---

If you don't want or can't use UTF-8 as character set for whatever reason, you will nevertheless be able to access the file. How does that work? When Cygwin converts the filename from UTF-16 to your character set, it recognizes characters which can't be converted. If that occurs, Cygwin replaces the non-convertible character with a special character sequence. The sequence starts with an ASCII CAN character (hex code 0x18, equivalent Control-X), followed by the UTF-8 representation of the character. The result is a filename containing some ugly looking characters. While it doesn't **look** nice, it **is** nice, because Cygwin knows how to convert this filename back to UTF-16. The filename will be converted using your usual character set. However, when Cygwin recognizes an ASCII CAN character, it skips over the ASCII CAN and handles the following bytes as a UTF-8 character. Thus, the filename is symmetrically converted back to UTF-16 and you can access the file.

---

**Note**

Please be aware that this method is not entirely foolproof. In some character set combinations it might not work for certain native characters.

Only by using the UTF-8 charset you can avoid this problem safely.

---

### 3.4.5 Case sensitive filenames

In the Win32 subsystem filenames are only case-preserved, but not case-sensitive. You can't access two files in the same directory which only differ by case, like `Abc` and `aBc`. While NTFS (and some remote filesystems) support case-sensitivity, the NT kernel starting with Windows XP does not support it by default. Rather, you have to tweak a registry setting and reboot. For that reason, case-sensitivity can not be supported by Cygwin, unless you change that registry value.

If you really want case-sensitivity in Cygwin, you can switch it on by setting the registry value

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\kernel\obcaseinsensitive
```

to 0 and reboot the machine. For least surprise, Cygwin expects this registry value also on Windows 2000, which usually doesn't know this registry key. If you want case-sensitivity on Windows 2000, just create that registry value and set it to 0. **Only** on Windows 2000 you don't have to reboot to bring it into effect, rather stopping all Cygwin processes and then restarting them is sufficient.

---

**Note**

When installing Microsoft's Services For Unix (SFU), you're asked if you want to use case-sensitive filenames. If you answer "yes" at this point, the installer will change the aforementioned registry value to 0, too. So, if you have SFU installed, there's some chance that the registry value is already set to case sensitivity.

---

After you set this registry value to 0, Cygwin will be case-sensitive by default on NTFS and NFS filesystems. However, there are limitations: while two **programs** `Abc.exe` and `aBc.exe` can be created and accessed like other files, starting applications is still case-insensitive due to Windows limitations and so the program you try to launch may not be the one actually started. Also, be aware that using two filenames which only differ by case might result in some weird interoperability issues with native Win32 applications. You're using case-sensitivity at your own risk. You have been warned!

Even if you use case-sensitivity, it might be feasible to switch to case-insensitivity for certain paths for better interoperability with native Win32 applications (even if it's just Windows Explorer). You can do this on a per-mount point base, by using the "posix=0" mount option in `/etc/fstab`, or your `/etc/fstab.d/$USER` file.

`/cygdrive` paths are case-insensitive by default. The reason is that the native Windows `%PATH%` environment variable is not always using the correct case for all paths in it. As a result, if you use case-sensitivity on the `/cygdrive` prefix, your shell might claim that it can't find Windows commands like `attrib` or `net`. To ease the pain, the `/cygdrive` path is case-insensitive by default and you have to use the "posix=1" setting explicitly in `/etc/fstab` or `/etc/fstab.d/$USER` to switch it to case-sensitivity, or you have to make sure that the native Win32 `%PATH%` environment variable is using the correct case for all paths throughout.

---

Note that mount points as well as device names and virtual paths like `/proc` are always case-sensitive! The only exception are the subdirectories and filenames under `/proc/registry`, `/proc/registry32` and `/proc/registry64`. Registry access is always case-insensitive. Read on for more information.

### 3.4.6 POSIX devices

While there is no need to create a POSIX `/dev` directory, the directory is automatically created as part of a Cygwin installation. It's existence is often a prerequisite to run certain applications which create symbolic links, fifos, or UNIX sockets in `/dev`. Also, the directories `/dev/shm` and `/dev/mqueue` are required to exist to use named POSIX semaphores, shared memory, and message queues, so a system without a real `/dev` directory is functionally crippled.

Apart from that, Cygwin automatically simulates POSIX devices internally. Up to Cygwin 1.7.11, these devices couldn't be seen with the command `ls /dev/` although commands such as `ls /dev/tty` worked fine. Starting with Cygwin 1.7.12, the `/dev` directory is automatically populated with existing POSIX devices by Cygwin in a way comparable with a `udev` based virtual `/dev` directory under Linux.

Cygwin supports the following character devices commonly found on POSIX systems:

```
/dev/null
/dev/zero
/dev/full

/dev/console Pseudo device name for the current console window of a session.
Up to Cygwin 1.7.9, this was the only name for a console.
Different consoles were indistinguishable.
Cygwin's /dev/console is not quite comparable with the console
device on UNIX machines.

/dev/cons0 Starting with Cygwin 1.7.10, Console sessions are numbered from
/dev/cons1 /dev/cons0 upwards. Console device names are pseudo device
... names, only accessible from processes within this very console
session. This is due to a restriction in Windows.

/dev/tty The current controlling tty of a session.

/dev/ptmx Pseudo tty master device.

/dev/pty0 Pseudo ttys are numbered from /dev/pty0 upwards as they are
/dev/pty1 requested.
...

/dev/ttyS0 Serial communication devices. ttyS0 == Win32 COM1,
/dev/ttyS1 ttyS1 == COM2, etc.
...

/dev/pipe
/dev/fifo

/dev/mem The physical memory of the machine. Note that access to the
/dev/port physical memory has been restricted with Windows Server 2003.
/dev/kmem Since this OS, you can't access physical memory from user space.

/dev/kmsg Kernel message pipe, for usage with sys logger services.

/dev/random Random number generator.
/dev/urandom

/dev/dsp Default sound device of the system.
```

Cygwin also has several Windows-specific devices:

```

/dev/com1 The serial ports, starting with COM1 which is the same as ttyS0.
/dev/com2 Please use /dev/ttySx instead.
...

/dev/conin Same as Windows CONIN$.
/dev/conout Same as Windows CONOUT$.
/dev/clipboard The Windows clipboard, text only
/dev/windows The Windows message queue.

```

Block devices are accessible by Cygwin processes using fixed POSIX device names. These POSIX device names are generated using a direct conversion from the POSIX namespace to the internal NT namespace. E.g. the first harddisk is the NT internal device `\device\harddisk0\partition0` or the first partition on the third harddisk is `\device\harddisk2\partition1`. The first floppy in the system is `\device\floppy0`, the first CD-ROM is `\device\cdrom0` and the first tape drive is `\device\tape0`.

The mapping from physical device to the name of the device in the internal NT namespace can be found in various places. For hard disks and CD/DVD drives, the Windows "Disk Management" utility (part of the "Computer Management" console) shows that the mapping of "Disk 0" is `\device\harddisk0`. "CD-ROM 2" is `\device\cdrom2`. Another place to find this mapping is the "Device Management" console. Disks have a "Location" number, tapes have a "Tape Symbolic Name", etc. Unfortunately, the places where this information is found is not very well-defined.

For external disks (USB-drives, CF-cards in a cardreader, etc) you can use Cygwin to show the mapping. `/proc/partitions` contains a list of raw drives known to Cygwin. The `df` command shows a list of drives and their respective sizes. If you match the information between `/proc/partitions` and the `df` output, you should be able to figure out which external drive corresponds to which raw disk device name.

---

#### Note

Apart from tape devices which are not block devices and are by default accessed directly, accessing mass storage devices raw is something you should only do if you know what you're doing and know how to handle the information. **Writing** to a raw mass storage device you should only do if you **really** know what you're doing and are aware of the fact that any mistake can destroy important information, for the device, and for you. So, please, handle this ability with care. **You have been warned.**

---

Last but not least, the mapping from POSIX `/dev` namespace to internal NT namespace is as follows:

POSIX device name	Internal NT device name
<code>/dev/st0</code>	<code>\device\tape0, rewind</code>
<code>/dev/nst0</code>	<code>\device\tape0, no-rewind</code>
<code>/dev/st1</code>	<code>\device\tape1</code>
<code>/dev/nst1</code>	<code>\device\tape1</code>
...	
<code>/dev/st15</code>	
<code>/dev/nst15</code>	
<code>/dev/fd0</code>	<code>\device\floppy0</code>
<code>/dev/fd1</code>	<code>\device\floppy1</code>
...	
<code>/dev/fd15</code>	
<code>/dev/sr0</code>	<code>\device\cdrom0</code>
<code>/dev/sr1</code>	<code>\device\cdrom1</code>
...	
<code>/dev/sr15</code>	
<code>/dev/scd0</code>	<code>\device\cdrom0</code>
<code>/dev/scd1</code>	<code>\device\cdrom1</code>
...	
<code>/dev/scd15</code>	
<code>/dev/sda</code>	<code>\device\harddisk0\partition0 (whole disk)</code>

---

```

/dev/sda1      \device\harddisk0\partition1  (first partition)
...
/dev/sda15     \device\harddisk0\partition15 (fifteenth partition)

/dev/sdb       \device\harddisk1\partition0
/dev/sdb1      \device\harddisk1\partition1

[up to]

/dev/sddx      \device\harddisk127\partition0
/dev/sddx1     \device\harddisk127\partition1
...
/dev/sddx15    \device\harddisk127\partition15

```

if you don't like these device names, feel free to create symbolic links as they are created on Linux systems for convenience:

```

ln -s /dev/sr0 /dev/cdrom
ln -s /dev/nst0 /dev/tape
...

```

### 3.4.7 The .exe extension

Win32 executable 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.

```

bash$ ls *
a      a.exe      b.exe
bash$ ls -i a a.exe
445885548 a      435996602 a.exe
bash$ 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 shell script has precedence and is selected for execution of **myprog**. Note that this was quite the reverse up to Cygwin 1.5.19. It has been changed for consistency with the rest of Cygwin.

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.

### 3.4.8 The /proc filesystem

Cygwin, like Linux and other similar operating systems, supports the `/proc` virtual filesystem. The files in this directory are representations of various aspects of your system, for example the command `cat /proc/cpuinfo` displays information such as what model and speed processor you have.

One unique aspect of the Cygwin `/proc` filesystem is `/proc/registry`, see next section.

The Cygwin `/proc` is not as complete as the one in Linux, but it provides significant capabilities. The `procps` package contains several utilities that use it.

### 3.4.9 The /proc/registry filesystem

The `/proc/registry` filesystem provides read-only access to the Windows registry. It displays each KEY as a directory and each VALUE as a file. As anytime you deal with the Windows registry, use caution since changes may result in an unstable

or broken system. There are additionally subdirectories called `/proc/registry32` and `/proc/registry64`. They are identical to `/proc/registry` on 32 bit host OSes. On 64 bit host OSes, `/proc/registry32` opens the 32 bit processes view on the registry, while `/proc/registry64` opens the 64 bit processes view.

Reserved characters (`'/'`, `'\'`, `':'`, and `'%'`) or reserved names (`.` and `..`) are converted by percent-encoding:

```
bash$ regtool list -v '\HKEY_LOCAL_MACHINE\SYSTEM\MountedDevices'
...
\DosDevices\C: (REG_BINARY) = cf a8 97 e8 00 08 fe f7
...
bash$ cd /proc/registry/HKEY_LOCAL_MACHINE/SYSTEM
bash$ ls -l MountedDevices
...
-r--r----- 1 Admin SYSTEM 12 Dec 10 11:20 %5CDosDevices%5CC%3A
...
bash$ od -t x1 MountedDevices/%5CDosDevices%5CC%3A
00000000 cf a8 97 e8 00 08 fe f7 01 00 00 00
```

The unnamed (default) value of a key can be accessed using the filename `@`.

If a registry key contains a subkey and a value with the same name `foo`, Cygwin displays the subkey as `foo` and the value as `foo%val`.

### 3.4.10 The @pathnames

To circumvent the limitations on shell line length in the native Windows command shells, Cygwin programs, when invoked by non-Cygwin processes, 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. In the following example compare the behaviors `/bin/echo` when run from `bash` and from the Windows command prompt.

---

#### Example 3.2 Using @pathname

---

```
bash$ /bin/echo 'This is "a long" line' > mylist
bash$ /bin/echo @mylist
@mylist
bash$ cmd
c:\> c:\cygwin\bin\echo @mylist
This is a long line
```

---

## 3.5 The CYGWIN environment variable

### 3.5.1 Implemented options

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) detect_blooda` - If set, Cygwin will try to detect foreign applications which try to inject threads into a Cygwin process, or which redirect system sockets by providing an enforced so-called `Layered Service Provider`. This may or may not help to detect **BLODAs**. Don't use this option for day-to-day usage, it will slow down every thread and socket creation!
  - `(no) dosfilewarning` - If set, Cygwin will warn the first time a user uses an "MS-DOS" style path name rather than a POSIX-style path name. Defaults to set.
  - `(no) export` - If set, the final values of these settings are re-exported to the environment as `CYGWIN` again. Defaults to off.
-

- `error_start:Win32filepath` - if set, runs `Win32filepath` when cygwin encounters a fatal error, which is useful for debugging. `Win32filepath` is usually set to the path to **gdb** or **dumper**, for example `C:\cygwin\bin\gdb.exe`. There is no default set.
- `(no)glob[:ignorecase]` - if set, command line arguments containing UNIX-style file wildcard characters (brackets, braces, question mark, asterisk, escaped with `\`) are expanded into lists of files that match those wildcards. This is applicable only to programs run from non-Cygwin programs such as a CMD prompt. That means that this setting does not affect globbing operations for shells such as `bash`, `sh`, `tcsh`, `zsh`, etc. Default is set.  
  
This option also accepts an optional `[no]ignorecase` modifier. If supplied, wildcard matching is case insensitive. The default is `noignorecase`.
- `(no)pipe_byte` - causes Cygwin to open pipes in byte mode rather than message mode.
- `proc_retry:n` - causes `fork()` and `exec*()` to retry `n` times when a child process fails due to certain windows-specific errors. These errors usually occur when processes are being started while a user is logging off.
- `(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.
- `(no)winsymlinks` - if set, Cygwin creates symlinks as Windows shortcuts with a special header and the R/O attribute set. If not set, Cygwin creates symlinks as plain files with a magic number, a path and the system attribute set. Defaults to not set since plain file symlinks are faster to write and faster to read.  
  
Please note that symlinks created under Cygwin 1.7 or later are not readable by older Cygwin releases because the new symlinks use UTF-16 to encode the target filename, while the old symlinks used the current ANSI or OEM charset.

### 3.5.2 Obsolete options

Certain CYGWIN options available in past releases have been removed in Cygwin 1.7 for one reason or another. These obsolete options are listed below.

- `(no)binmode` - This option has been removed because all file opens default to binary mode, unless the open mode has been specified explicitly in the `open(2)` call.
- `check_case` - This option has been removed in favor of real case sensitivity and the per-mount option "`posix=[0|1]`". For more information, read the documentation in Section 3.1.2 and Section 3.4.5.
- `codepage:[ansi|oem]` - This option controlled which character set is used for file and console operations. Since Cygwin is now doing all character conversion by itself, depending on the application call to the `setlocale()` function, and in turn by the setting of the environment variables `$LANG`, `$LC_ALL`, or `$LC_CTYPE`, this setting became superfluous.
- `(no)envcache` - Originally, `envcache` controlled caching of environment variable conversion between Win32 and POSIX. The default setting works fine, the option was just useless.
- `forkchunk:[intval]` - This option allowed to influence the `fork()` function in the way the memory of the parent process gets copied to the child process. This functionality was only useful for Windows 95/98/Me.
- `(no)ntea` - This option has been removed since it only fakes security which is considered dangerous and useless. It also created an uncontrollably large file on FAT and was entirely useless on FAT32.
- `(no)ntsec` - This option has been removed in favor of the per-mount option "`acl"/"noacl`". For more information, read the documentation in Section 3.1.2.
- `(no)server` - Originally this option had to be enabled on the client side to use features only available when running **cygserver**. This option has been removed because Cygwin now always tries to contact **cygserver** if a function is called which requires **cygserver** being available. For more information, read the documentation in Section 3.7.
- `(no)smbntsec` - This option has been removed in favor of the per-mount option "`acl"/"noacl`". For more information, read the documentation in Section 3.1.2.
- `(no)strip_title` - Removed because setting the Window title can be controlled by the application via Escape sequences.

- `(no)title` - Removed because setting the Window title can be controlled by the application via Escape sequences.
- `(no)transparent_exe` - This option has been removed because the behaviour it switched on is now the standard behaviour in Cygwin.
- `(no)traverse` - This option has been removed because traverse checking is not quite correctly implemented by Microsoft and it's behaviour has been getting worse with each new OS version. This complicates its usage so the option has been removed for now.
- `(no)tty` - If set, Cygwin enabled extra support (i.e., `termios`) for UNIX-like ttys in the Windows console. This option has been removed because it can be easily replaced by using a terminal like **mintty**, and it does not work well with some Windows programs.
- `(no)upcaseenv` - This option could be used to convert all environment variables to uppercase. This was the default behavior in releases prior to Cygwin 1.7. Since keeping the case of environment variables intact is POSIXly correct, Cygwin now does not change the case of environment variables, except for a restricted set to maintain minimal backward compatibility. The current list of always uppercased variables is:

```
ALLUSERSPROFILE
COMMONPROGRAMFILES
COMPUTERNAME
COMSPEC
HOME
HOMEDRIVE
HOMEPATH
NUMBER_OF_PROCESSORS
OS
PATH
PATHEXT
PROCESSOR_ARCHITECTURE
PROCESSOR_IDENTIFIER
PROCESSOR_LEVEL
PROCESSOR_REVISION
PROGRAMFILES
SYSTEMDRIVE
SYSTEMROOT
TEMP
TERM
TMP
TMPDIR
WINDIR
```

## 3.6 Using Windows security in Cygwin

This section discusses how the Windows security model is utilized in Cygwin to implement POSIX-like permissions, as well as how the Windows authentication model is used to allow cygwin applications to switch users in a POSIX-like fashion.

The setting of POSIX-like file and directory permissions is controlled by the **mount** option `(no)acl` which is set to `acl` by default.

We start with a short overview. Note that this overview must be necessarily short. If you want to learn more about the Windows security model, see the **Access Control** article in MSDN documentation.

POSIX concepts and in particular the POSIX security model are not discussed here, but assumed to be understood by the reader. If you don't know the POSIX security model, search the web for beginner documentation.

### 3.6.1 Overview

In the Windows security model, almost any "object" is securable. "Objects" are files, processes, threads, semaphores, etc.

Every object has a data structure attached, called a "security descriptor" (SD). The SD contains all information necessary to control who can access an object, and to determine what they are allowed to do to or with it. The SD of an object consists of five parts:

- Flags which control several aspects of this SD. This is not discussed here.
- The SID of the object owner.
- The SID of the object owner group.
- A list of "Access Control Entries" (ACE), called the "Discretionary Access Control List" (DACL).
- Another list of ACEs, called the "Security Access Control List" (SACL), which doesn't matter for our purpose. We ignore it here.

Every ACE contains a so-called "Security Identifier" (SID) and other stuff which is explained a bit later. Let's talk about the SID first.

A SID is a unique identifier for users, groups, computers and Active Directory (AD) domains. SIDs are basically comparable to POSIX user ids (UIDs) and group ids (GIDs), but are more complicated because they are unique across multiple machines or domains. A SID is a structure of multiple numerical values. There's a convenient convention to type SIDs, as a string of numerical fields separated by hyphen characters. Here's an example:

SID of a machine "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 first field is always "S", which is just a notational convention to show that this is a SID. The second field is the version number of the SID structure, so far there exists only one version of SIDs, so this field is always 1. The third and fourth fields represent the "authority" which can be thought of as a type or category of SIDs. There are a couple of builtin accounts and accounts with very special meaning which have certain well known values in these third and fourth fields. However, computer and domain SIDs always start with "S-1-5-21". The next three fields, all 32 bit values, represent the unique 96 bit identifier of the computer system. This is a hopefully unique value all over the world, but in practice it's sufficient if the computer SIDs are unique within a single Windows network.

As you can see in the above example, SIDs of users (and groups) are identical to the computer SID, except for an additional part, the so-called "relative identifier" (RID). So the SID of a user is always uniquely attached to the system on which the account has been generated.

It's a bit different in domains. The domain has its own SID, and that SID is identical to the SID of the first domain controller, on which the domain is created. Domain user SIDs look exactly like the computer user SIDs, the leading part is just the domain SID and the RID is created when the user is created.

Ok, consider you created a new domain "bar" on some new domain controller and you would like to create a domain account "johndoe":

SID of a domain "bar.local":

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

SID of a user "johndoe" in the domain "bar.local":

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

So you now have two accounts called johndoe, one account created on the machine "foo", one created in the domain "bar.local". Both have different SIDs and not even the RID is the same. How do the systems know it's the same account? After all, the name is the same, right? The answer is, these accounts are **not** identical. All machines on the network will treat these SIDs as identifying two separate accounts. One is "FOO\johndoe", the other one is "BAR\johndoe" or "johndoe@bar.local". Different SID, different account. Full stop.



The last part of the SID, the so called "Relative Identifier" (RID), is by default used as UID and/or GID under Cygwin when you create the `/etc/passwd` and `/etc/group` files using the **mkpasswd** and **mkgroup** tools. Domain account UIDs and GIDs are offset by 10000 by default which might be a bit low for very big organizations. Fortunately there's an option in both tools to change the offset...

Do you still remember the SIDs with special meaning? In official notation they are called "well-known SIDs". For example, POSIX has no GID for the group of "all users" or "world" or "others". The last three rwx bits in a unix-style permission value just represent the permissions for "everyone who is not the owner or is member of the owning group". Windows has a SID for these poor souls, the "Everyone" SID. Other well-known SIDs represent circumstances under which a process is running, rather than actual users or groups. Here are a few examples for well-known SIDs:

Everyone	S-1-1-0	Simply everyone...
Batch	S-1-5-3	Processes started via the task scheduler are member of this group.
Interactive	S-1-5-4	Only processes of users which are logged in via an interactive session are members here.
Authenticated Users	S-1-5-11	Users which have gone through the authentication process and survived. Anonymously accessing users are not included here.
SYSTEM	S-1-5-18	A special account which has all kinds of dangerous rights, sort of an uber-root account.

For a full list please refer to the MSDN document [Well-known SIDs](#). The Cygwin package called "csih" provides a tool, `/usr/lib/csih/getAccountName.exe`, which can be used to print the (possibly localized) name for the various well-known SIDs.

Naturally, well-known SIDs are the same on each machine, so they are not unique to a machine or domain. They have the same meaning across the Windows network.

Additionally, there are a couple of well-known builtin groups, which have the same SID on every machine and which have certain user rights by default:

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

For instance, every account is usually member in the "Users" group. All administrator accounts are member of the "Administrators" group. That's all about it as far as single machines are involved. In a domain environment it's a bit more tricky. Since these SIDs are not unique to a machine, every domain user and every domain group can be a member of these well known groups. Consider the domain group "Domain Admins". This group is by default in the "Administrators" group. Let's assume the above computer called "foo" is a member machine of the domain "bar.local". If you stick the user "BAR\johndoe" into the group "Domain Admins", this guy will automatically be a member of the administrators group on "foo" when logging on to "foo". Neat, isn't it?

Back to ACE and ACL. POSIX is able to create three different permissions, the permissions for the owner, for the group and for the world. In contrast the Windows ACL has a potentially infinite number of members... as long as they fit into 64K. Every member is an ACE. ACE consist of three parts:

- The type of the ACE (allow ACE or deny ACE).
- Permission bits, 32 of them.
- The SID for which the permissions are allowed or denied.

The two (for us) important types of ACEs are the "access allowed ACE" and the "access denied ACE". As the names imply, the allow ACE tells the system to allow the given permissions to the SID, the deny ACE results in denying the specific permission bits.

The possible permissions on objects are more detailed than in POSIX. For example, the permission to delete an object is different from the permission to change object data, and even changing object data can be separated into different permission bits for

different kind of data. But there's a problem with the definition of a "correct" ACL which disallows mapping of certain POSIX permissions cleanly. See Section 3.6.4.

POSIX is able to create only three different permissions? Not quite. Newer operating systems and file systems on POSIX systems also provide access control lists. Two different APIs exist for accessing these ACLs, the Solaris API and the POSIX API. Cygwin implements the Solaris API to access Windows ACLs in a Unixy way. At the time of writing this document, the Cygwin implementation of the Solaris API isn't quite up to speed. For instance, it doesn't handle access denied ACEs gracefully. So, use with care. Online man pages for the Solaris ACL API can be found on <http://docs.sun.com>.

### 3.6.2 File permissions

On NTFS and if the `noacl` mount option is not specified for a mount point, Cygwin sets file permissions as in POSIX. Basically this is done by defining a SD with the matching owner and group SIDs, and a DACL which contains ACEs for the owner, the group and for "Everyone", which represents what POSIX calls "others".

To use Windows security correctly, Cygwin depends on the files `/etc/passwd` and `/etc/group`. These files define the translation between the Cygwin uid/gid and the Windows SID. The SID is stored in the `pw_gecos` field in `/etc/passwd`, and in the `gr_passwd` field in `/etc/group`. Since the `pw_gecos` field can contain more information than just a SID, there are some rules for the layout. It's required that the SID is the last entry of the `pw_gecos` field, assuming that the entries in `pw_gecos` are comma-separated. The commands **mkpasswd** and **mkgroup** usually do this for you.

Another interesting entry in the `pw_gecos` field (which is also usually created by running **mkpasswd**) is the Windows user name entry. It takes the form "U-domain\username" and is sometimes used by services to authenticate a user. Logging in through **telnet** is a common scenario.

A typical snippet from `/etc/passwd`:

---

#### Example 3.3 /etc/passwd:

```
SYSTEM*:18:544:,S-1-5-18::
Administrators*:544:544:,S-1-5-32-544::
Administrator:unused:500:513:U-FOO\Administrator,S ←
-1-5-21-790525478-115176313-839522115-500:/home/Administrator:/bin/bash
corinna:unused:11001:11125:U-BAR\corinna,S-1-5-21-2913048732-1697188782-3448811101-1001:/ ←
home/corinna:/bin/tcsh
```

---

The `SYSTEM` entry is usually needed by services. The `Administrators` entry (Huh? A group in `/etc/passwd`?) is only here to allow **ls** and similar commands to print some file ownerships correctly. Windows doesn't care if the owner of a file is a user or a group. In older versions of Windows NT the default ownership for files created by an administrator account was set to the group `Administrators` instead of to the creating user account. This has changed, but you can still switch to this setting on newer systems. So it's convenient to have the `Administrators` group in `/etc/passwd`.

The really interesting entries are the next two. The `Administrator` entry is for the local administrator, the `corinna` entry matches the `corinna` account in the domain `BAR`. The information given in the `pw_gecos` field are all we need to exactly identify an account, and to have a two way translation, from Windows account name/SID to Cygwin account name uid and vice versa. Having this complete information allows us to choose a Cygwin user name and uid which doesn't have to match the Windows account at all. As long as the `pw_gecos` information is available, we're on the safe side:

---

#### Example 3.4 /etc/passwd, tweaked:

```
root:unused:0:513:U-FOO\Administrator,S-1-5-21-790525478-115176313-839522115-500:/home/ ←
Administrator:/bin/bash
thursday_next:unused:11001:11125:U-BAR\corinna,S ←
-1-5-21-2913048732-1697188782-3448811101-1001:/home/corinna:/bin/tcsh
```

---

The above `/etc/passwd` will still work fine. You can now login via **ssh** as the user "root", and Cygwin dutifully translates "root" into the Windows user "FOO\Administrator" and files owned by `FOO\Administrator` are shown to have the uid 0 when

---

calling **ls -ln**. All you do you're actually doing as Administrator. Files created as root will be owned by FOO\Administrator. And the domain user BAR\corinna can now happily pretend to be Thursday Next, but will wake up sooner or later finding out she's still actually the domain user BAR\corinna...

Do I have to mention that you can also rename groups in `/etc/group`? As long as the SID is present and correct, all is well. This allows you to, for instance, rename the "Administrators" group to "root" as well:

---

**Example 3.5** `/etc/group`, tweaked:

---

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

---

Last but not least, you can also change the primary group of a user in `/etc/passwd`. The only requirement is that the user is actually a member of the new primary group in Windows. For instance, normal users in a domain environment are members in the group "Domain Users", which in turn belongs to the well-known group "Users". So, if it's more convenient in your environment for the user's primary group to be "Users", just set the user's primary group in `/etc/passwd` to the Cygwin uid of "Users" (see in `/etc/group`, default 545) and let the user create files with a default group ownership of "Users".

---

#### Note

If you wish to make these kind of changes to `/etc/passwd` and `/etc/group`, do so only if you feel comfortable with the concepts. Otherwise, do not be surprised if things break in either subtle or surprising ways! If you do screw things up, revert to copies of `/etc/passwd` and `/etc/group` files created by **mkpasswd** and **mkgroup**. (Make backup copies of these files before modifying them.) Especially, don't change the UID or the name of the user SYSTEM. It may mostly work, but some Cygwin applications running as a local service under that account could suddenly start behaving strangely.

---

### 3.6.3 Special values of user and group ids

If the current user is not present in `/etc/passwd`, that user's uid is set to a special value of 400. The user name for the current user will always be shown correctly. If another user (or a Windows group, treated as a user) is not present in `/etc/passwd`, the uid of that user will have a special value of -1 (which would be shown by **ls** as 65535). The user name shown in this case will be '????????'.

If the current user is not present in `/etc/passwd`, that user's login gid is set to a special value of 401. The gid 401 is shown as 'mkpasswd', indicating the command that should be run to alleviate the situation.

If another user is not present in `/etc/passwd`, that user's login gid is set to a special value of -1. If the user is present in `/etc/passwd`, but that user's group is not in `/etc/group` and is not the login group of that user, the gid is set to a special value of -1. The name of this group (id -1) will be shown as '????????'.

If the current user is present in `/etc/passwd`, but that user's login group is not present in `/etc/group`, the group name will be shown as 'mkgroup', again indicating the appropriate command.

A special case is if the current user's primary group SID is noted in the user's `/etc/passwd` entry using another group id than the group entry of the same group SID in `/etc/group`. This should be noted and corrected. The group name printed in this case is 'passwd/group\_GID\_clash(PPP/GGG)', with PPP being the gid as noted in `/etc/passwd` and GGG the gid as noted in `/etc/group`.

To summarize:

- If the current user doesn't show up in `/etc/passwd`, it's *group* will be named 'mkpasswd'.
- Otherwise, if the login group of the current user isn't in `/etc/group`, it will be named 'mkgroup'.
- Otherwise a group not in `/etc/group` will be shown as '????????' and a user not in `/etc/passwd` will be shown as "????????".
- If different group ids are used for a group with the same SID, the group name is shown as 'passwd/group\_GID\_clash(PPP/GGG)' with PPP and GGG being the different group ids.

Note that, since the special user and group names are just indicators, nothing prevents you from actually having a user named 'mkpasswd' in `/etc/passwd` (or a group named 'mkgroup' in `/etc/group`). If you do that, however, be aware of the possible confusion.

---

### 3.6.4 The POSIX permission mapping leak

As promised earlier, here's the problem when trying to map the POSIX permission model onto the Windows permission model. There's a leak in the definition of a "correct" ACL which disallows a certain POSIX permission setting. The official documentation explains in short the following:

- The requested permissions are checked against all ACEs of the user as well as all groups the user is member of. The permissions given in these user and groups access allowed ACEs are accumulated and the resulting set is the set of permissions of that user given for that object.
- The order of ACEs is important. The system reads them in sequence until either any single requested permission is denied or all requested permissions are granted. Reading stops when this condition is met. Later ACEs are not taken into account.
- All access denied ACEs **should** precede any access allowed ACE. ACLs following this rule are called "canonical"

Note that the last rule is a preference or a definition of correctness. It's not an absolute requirement. All Windows kernels will correctly deal with the ACL regardless of the order of allow and deny ACEs. The second rule is not modified to get the ACEs in the preferred order.

Unfortunately the security tab in the file properties dialog of the Windows Explorer insists to rearrange the order of the ACEs to canonical order before you can read them. Thank God, the sort order remains unchanged if one presses the Cancel button. But don't even **think** of pressing OK...

Canonical ACLs are unable to reflect each possible combination of POSIX permissions. Example:

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

Ok, so here's the first try to create a matching ACL, assuming the Windows permissions only have three bits, as their POSIX counterpart:

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

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

Second 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.

Third 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, either. How should this problem be solved? According to the canonical order 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 on all existing versions of Windows NT, at the time of writing from at least Windows 2000 up to Server 2008 R2. Only the GUIs aren't able (or willing) to deal with that order.

### 3.6.5 Switching the user context

Since Windows XP, Windows users have been accustomed to the "Switch User" feature, which switches the entire desktop to another user while leaving the original user's desktop "suspended". Another Windows feature (since Windows 2000) is the "Run as..." context menu entry, which allows you to start an application using another user account when right-clicking on applications and shortcuts.

On POSIX systems, this operation can be performed by processes running under the privileged user accounts (usually the "root" user account) on a per-process basis. This is called "switching the user context" for that process, and is performed using the POSIX **setuid** and **seteuid** system calls.

While this sort of feature is available on Windows as well, Windows does not support the concept of these calls in a simple fashion. Switching the user context in Windows is generally a tricky process with lots of "behind the scenes" magic involved.

Windows uses so-called 'access tokens' to identify a user and its permissions. Usually the access token is created at logon time and then it's attached to the starting process. Every new process within a session inherits the access token from its parent process. Every thread can get its own access token, which allows, for instance, to define threads with restricted permissions.

### 3.6.6 Switching the user context with password authentication

To switch the user context, the process has to request such an access token for the new user. This is typically done by calling the Win32 API function **LogonUser** with the user name and the user's cleartext password as arguments. If the user exists and the password was specified correctly, the access token is returned and either used in **ImpersonateLoggedOnUser** to change the user context of the current thread, or in **CreateProcessAsUser** to change the user context of a spawned child process.

Later versions of Windows define new functions in this context and there are also functions to manipulate existing access tokens (usually only to restrict them). Windows Vista also adds subtokens which are attached to other access tokens which plays an important role in the UAC (User Access Control) facility of Vista and later. However, none of these extensions to the original concept are important for this documentation.

Back to this logon with password, how can this be used to implement **set(e)uid**? Well, it requires modification of the calling application. Two Cygwin functions have been introduced to support porting **setuid** applications which only require login with passwords. You only give Cygwin the right access token and then you can call **seteuid** or **setuid** as usual in POSIX applications. Porting such a **setuid** application is illustrated by a short example:

```
/* First include all needed cygwin stuff. */
#ifdef __CYGWIN__
#include <windows.h>
#include <sys/cygwin.h>
#endif

[...]

struct passwd *user_pwd_entry = getpwnam (username);
char *cleartext_password = getpass ("Password:");

[...]

#ifdef __CYGWIN__
/* Patch the typical password test. */
{
    HANDLE token;

    /* Try to get the access token from Windows. */
    token = cygwin_logon_user (user_pwd_entry, cleartext_password);
    if (token == INVALID_HANDLE_VALUE)
        error_exit;
    /* Inform Cygwin about the new impersonation token. */
    cygwin_set_impersonation_token (token);
    /* Cygwin is now able, to switch to that user context by setuid or seteuid calls. */
}
```

```
    }
#else
    /* Use standard method on non-Cygwin systems. */
    hashed_password = crypt (cleartext_password, salt);
    if (!user_pwd_entry ||
        strcmp (hashed_password, user_pwd_entry->pw_password))
        error_exit;
#endif /* CYGWIN */

[...]
```

```
/* Everything else remains the same! */

setegid (user_pwd_entry->pw_gid);
seteuid (user_pwd_entry->pw_uid);
execl ("/bin/sh", ...);
```

### 3.6.7 Switching the user context without password, Method 1: Create a token from scratch

An unfortunate aspect of the implementation of **set(e)uid** is the fact that the calling process requires the password of the user to which to switch. Applications such as **sshd** wishing to switch the user context after a successful public key authentication, or the **cron** application which, again, wants to switch the user without any authentication are stuck here. But there are other ways to get new user tokens.

One way is just to create a user token from scratch. This is accomplished by using an (officially undocumented) function on the NT function level. The NT function level is used to implement the Win32 level, and, as such is closer to the kernel than the Win32 level. The function of interest, **NtCreateToken**, allows you to specify user, groups, permissions and almost everything you need to create a user token, without the need to specify the user password. The only restriction for using this function is that the calling process needs the "Create a token object" user right, which only the SYSTEM user account has by default, and which is considered the most dangerous right a user can have on Windows systems.

That sounds good. We just start the servers which have to switch the user context (**sshd**, **inetd**, **cron**, ...) as Windows services under the SYSTEM (or LocalSystem in the GUI) account and everything just works. Unfortunately that's too simple. Using **NtCreateToken** has a few drawbacks.

First of all, beginning with Windows Server 2003, the permission "Create a token object" gets explicitly removed from the SYSTEM user's access token, when starting services under that account. That requires us to create a new account with this specific permission just to run this kind of services. But that's a minor problem.

A more important problem is that using **NtCreateToken** is not sufficient to create a new logon session for the new user. What does that mean? Every logon usually creates a new logon session. A logon session has a couple of attributes which are unique to the session. One of these attributes is the fact, that Windows functions identify the user domain and user name not by the SID of the access token owner, but only by the logon session the process is running under.

This has the following unfortunate consequence. Consider a service started under the SYSTEM account (up to Windows XP) switches the user context to DOMAIN\my\_user using a token created directly by calling the **NtCreateToken** function. A process running under this new access token might want to know under which user account it's running. The corresponding SID is returned correctly, for instance S-1-5-21-1234-5678-9012-77777. However, if the same process asks the OS for the user name of this SID something weird happens. For instance, the **LookupAccountSid** function will not return "DOMAIN\my\_user", but "NT AUTHORITY\SYSTEM" as the user name.

You might ask "So what?" After all, this only **looks** bad, but functionality and permission-wise everything should be ok. And Cygwin knows about this shortcoming so it will return the correct Cygwin username when asked. Unfortunately this is more complicated. Some native, non-Cygwin Windows applications will misbehave badly in this situation. A well-known example are certain versions of Visual-C++.

Last but not least, you don't have the usual comfortable access to network shares. The reason is that the token has been created without knowing the password. The password are your credentials necessary for network access. Thus, if you logon with a

password, the password is stored hidden as "token credentials" within the access token and used as default logon to access network resources. Since these credentials are missing from the token created with **NtCreateToken**, you only can access network shares from the new user's process tree by using explicit authentication, on the command line for instance:

```
bash$ net use '\\server\share' /user:DOMAIN\my_user my_users_password
```

Note that, on some systems, you can't even define a drive letter to access the share, and under some circumstances the drive letter you choose collides with a drive letter already used in another session. Therefore it's better to get used to accessing these shares using the UNC path as in

```
bash$ grep foo //server/share/foofile
```

### 3.6.8 Switching the user context without password, Method 2: LSA authentication package

We're looking for another way to switch the user context without having to provide the password. Another technique is to create an LSA authentication package. LSA is an acronym for "Local Security Authority" which is a protected part of the operating system which only allows changes to become active when rebooting the system after the change. Also, as soon as the LSA encounters serious problems (for instance, one of the protected LSA processes died), it triggers a system reboot. LSA is the part of the OS which cares for the user logons and which also creates logon sessions.

An LSA authentication package is a DLL which has to be installed as part of the LSA. This is done by tweaking a special registry key. Cygwin provides such an authentication package. It has to be installed and the machine has to be rebooted to activate it. This is the job of the shell script `/usr/bin/cyglsa-config` which is part of the Cygwin package.

After running `/usr/bin/cyglsa-config` and rebooting the system, the LSA authentication package is used by Cygwin when **set(e)uid** is called by an application. The created access token using this method has its own logon session.

This method has two advantages over the **NtCreateToken** method.

The very special and very dangerous "Create a token object" user right is not required by a user using this method. Other privileged user rights are still necessary, especially the "Act as part of the operating system" right, but that's just business as usual.

The user is correctly identified, even by delicate native applications which choke on that using the **NtCreateToken** method.

Disadvantages? Yes, sure, this is Windows. The access token created using LSA authentication still lacks the credentials for network access. After all, there still hasn't been any password authentication involved. The requirement to reboot after every installation or deinstallation of the cygwin LSA authentication DLL is just a minor inconvenience compared to that...

Nevertheless, this is already a lot better than what we get by using **NtCreateToken**, isn't it?

### 3.6.9 Switching the user context without password, Method 3: With password

Ok, so we have solved almost any problem, except for the network access problem. Not being able to access network shares without having to specify a cleartext password on the command line or in a script is a harsh problem for automated logons for testing purposes and similar stuff.

Fortunately there is a solution, but it has its own drawbacks. But, first things first, how does it work? The title of this section says it all. Instead of trying to logon without password, we just logon with password. The password gets stored two-way encrypted in a hidden, obfuscated area of the registry, the LSA private registry area. This part of the registry contains, for instance, the passwords of the Windows services which run under some non-default user account.

So what we do is to utilize this registry area for the purpose of **set(e)uid**. The Cygwin command **passwd -R** allows a user to specify his/her password for storage in this registry area. When this user tries to login using ssh with public key authentication, Cygwin's **set(e)uid** examines the LSA private registry area and searches for a Cygwin specific key which contains the password. If it finds it, it calls **LogonUser** under the hood, using this password. If that works, **LogonUser** returns an access token with all credentials necessary for network access.

For good measure, and since this way to implement **set(e)uid** is not only used by Cygwin but also by Microsoft's SFU (Services for Unix), we also look for a key stored by SFU (using the SFU command **regpwd**) and use that if it's available.

We got it. A full access token with its own logon session, with all network credentials. Hmm, that's heaven...

Back on earth, what about the drawbacks?

First, adding a password to the LSA private registry area requires administrative access. So calling **passwd -R** as a normal user will fail! Cygwin provides a workaround for this. If **cygserver** is started as a service running under the SYSTEM account (which is the default way to run **cygserver**) you can use **passwd -R** as normal, non-privileged user as well.

Second, as aforementioned, the password is two-way encrypted in a hidden, obfuscated registry area. Only SYSTEM has access to this area for listing purposes, so, even as an administrator, you can't examine this area with regedit. Right? No. Every administrator can start regedit as SYSTEM user:

```
bash$ date
Tue Dec  2 16:28:03 CET 2008
bash$ at 16:29 /interactive regedit.exe
```

Additionally, if an administrator knows under which name the private key is stored (which is well-known since the algorithms used to create the Cygwin and SFU keys are no secret), every administrator can access the password of all keys stored this way in the registry.

Conclusion: If your system is used exclusively by you, and if you're also the only administrator of your system, and if your system is adequately locked down to prevent malicious access, you can safely use this method. If your machine is part of a network which has dedicated administrators, and you're not one of these administrators, but you (think you) can trust your administrators, you can probably safely use this method.

In all other cases, don't use this method. You have been warned.

### 3.6.10 Switching the user context, how does it all fit together?

Now we learned about four different ways to switch the user context using the **set(e)uid** system call, but how does **set(e)uid** really work? Which method does it use now?

The answer is, all four of them. So here's a brief overview what **set(e)uid** does under the hood:

- When **set(e)uid** is called, it tests if the user context had been switched by an earlier call already, and if the new user account is the privileged user account under which the process had been started originally. If so, it just switches to the original access token of the process it had been started with.
- Next, it tests if an access token has been stored by an earlier call to **cygwin\_set\_impersonation\_token**. If so, it tests if that token matches the requested user account. If so, the stored token is used for the user context switch.  
If not, there's no predefined token which can just be used for the user context switch, so we have to create a new token. The order is as follows.
- Check if the user has stored the logon password in the LSA private registry area, either under a Cygwin key, or under a SFU key. If so, use this to call **LogonUser**. If this succeeds, we use the resulting token for the user context switch.
- Otherwise, check if the Cygwin-specific LSA authentication package has been installed and is functional. If so, use the appropriate LSA calls to communicate with the Cygwin LSA authentication package and use the returned token.
- Last chance, try to use the **NtCreateToken** call to create a token. If that works, use this token.
- If all of the above fails, our process has insufficient privileges to switch the user context at all, so **set(e)uid** fails and returns -1, setting **errno** to **EPERM**.

## 3.7 Cygserver

### 3.7.1 What is Cygserver?

Cygserver is a program which is designed to run as a background service. It provides Cygwin applications with services which require security arbitration or which need to persist while no other cygwin application is running.

The implemented services so far are:



- XSI IPC Message Queues.
- XSI IPC Semaphores.
- XSI IPC Shared Memory.
- Allows non-privileged users to store obfuscated passwords in the registry to be used by **setuid** and **seteuid** calls to create user tokens with network credentials. This service is used by **passwd -R**. Using the stored passwords in **set(e)uid** does not require running Cygserver. For details, see Section 3.6.5.
- This functionality is no longer used since Cygwin 1.7.6, but the interface is still available: Control slave tty/pty handle dispersal from tty owner to other processes without compromising the owner processes' security. Starting with Cygwin 1.7.6 another safe mechanism to share tty/pty handles is used.

### 3.7.2 Cygserver command line options

Options to Cygserver take the normal UNIX-style `-X` or `--longoption` form. Nearly all options have a counterpart in the configuration file (see below) so setting them on the command line isn't really necessary. Command line options override settings from the Cygserver configuration file.

The one-character options are prepended by a single dash, the long variants are prepended with two dashes. Arguments to options are marked in angle brackets below. These are not part of the actual syntax but are used only to denote the arguments. Note that all arguments are required. Cygserver has no options with optional arguments.

The recognized options are:

- `-f, --config-file <file>`

Use `<file>` as configuration file instead of the default configuration line. The default configuration file is `/etc/cygserver.conf`. The `--help` and `--version` options will print the default configuration pathname.

This option has no counterpart in the configuration file, for obvious reasons.

- `-c, --cleanup-threads <num>`

Number of threads started to perform cleanup tasks. Default is 2. Configuration file option: `kern.srv.cleanup_threads`

- `-r, --request-threads <num>`

Number of threads started to serve application requests. Default is 10. The `-c` and `-r` options can be used to play with Cygserver's performance under heavy load conditions or on slow machines. Configuration file option: `kern.srv.request_threads`

- `-d, --debug`

Log debug messages to `stderr`. These will clutter your `stderr` output with a lot of information, typically only useful to developers.

- `-e, --stderr`

Force logging to `stderr`. This is the default if `stderr` is connected to a tty. Otherwise, the default is logging to the system log. By using the `-e`, `-E`, `-y`, `-Y` options (or the appropriate settings in the configuration file), you can explicitly set the logging output as you like, even to both, `stderr` and `syslog`. Configuration file option: `kern.log.stderr`

- `-E, --no-stderr`

Don't log to `stderr`. Configuration file option: `kern.log.stderr`

- `-y, --syslog`

Force logging to the system log. This is the default, if `stderr` is not connected to a tty, e. g. redirected to a file. Configuration file option: `kern.log.syslog`

- `-Y, --no-syslog`

Don't log to syslog. Configuration file option: kern.log.syslog

- `-l, --log-level <level>`

Set the verbosity level of the logging output. Valid values are between 1 and 7. The default level is 6, which is relatively chatty. If you set it to 1, you will get only messages which are printed under severe conditions, which will result in stopping Cygserver itself. Configuration file option: kern.log.level

- `-m, --no-sharedmem`

Don't start XSI IPC Shared Memory support. If you don't need XSI IPC Shared Memory support, you can switch it off here. Configuration file option: kern.srv.sharedmem

- `-q, --no-msgqueues`

Don't start XSI IPC Message Queues. Configuration file option: kern.srv.msgqueues

- `-s, --no-semaphores`

Don't start XSI IPC Semaphores. Configuration file option: kern.srv.semaphores

- `-S, --shutdown`

Shutdown a running daemon and exit. Other methods are sending a SIGHUP to the Cygserver PID or, if running as service, calling ``net stop cygserver`` or ``cygrunsrv -E cygserver``.

- `-h, --help`

Output usage information and exit.

- `-V, --version`

Output version information and exit.

### 3.7.3 How to start Cygserver

Before you run Cygserver for the first time, you should run the `/usr/bin/cygserver-config` script once. It creates the default configuration file and, upon request, installs Cygserver as service. The script only performs a default install, with no further options given to Cygserver when running as service. Due to the wide configurability by changing the configuration file, that's typically not necessary.

You should always run Cygserver as a service under LocalSystem account. This is the way it is installed for you by the `/usr/bin/cygserver-config` script.

### 3.7.4 The Cygserver configuration file

Cygserver has many options, which allow you to customize the server to your needs. Customization is accomplished by editing the configuration file, which is by default `/etc/cygserver.conf`. This file is only read once, at startup of Cygserver. There's no option to re-read the file at runtime by, say, sending a signal to Cygserver.

The configuration file determines how Cygserver operates. There are options which set the number of threads running in parallel, options for setting how and what to log and options to set various maximum values for the IPC services.

The default configuration file delivered with Cygserver is installed to `/etc/defaults/etc`. The `/usr/bin/cygserver-config` script copies it to `/etc`, giving you the option to overwrite an already existing file or to leave it alone. Therefore, the `/etc` file is safe to be changed by you, since it will not be overwritten by a later update installation.

The default configuration file contains many comments which describe everything needed to understand the settings. A comment at the start of the file describes the syntax rules for the file. The default options are shown in the file but are commented out.

It is generally a good idea to uncomment only options which you intend to change from the default values. Since reading the options file on Cygserver startup doesn't take much time, it's also considered good practice to keep all other comments in the file. This keeps you from searching for clues in other sources.

## 3.8 Using Cygwin effectively with Windows

Cygwin is not a full operating system, and so must rely on Windows for accomplishing some tasks. For example, Cygwin provides a POSIX view of the Windows filesystem, but does not provide filesystem drivers of its own. Therefore part of using Cygwin effectively is learning to use Windows effectively. Many Windows utilities provide a good way to interact with Cygwin's predominately command-line environment. For example, **ipconfig.exe** provides information about network configuration, and **net.exe** views and configures network file and printer resources. Most of these tools support the `/?` switch to display usage information.

Unfortunately, no standard set of tools included with all versions of Windows exists. If you are unfamiliar with the tools available on your system, here is a general guide. Windows 2000 has only a basic set of tools, which later versions of Windows expanded. Microsoft also provides free downloads for Windows 2000 (the Resource Kit Tools), and XP (the Windows Support Tools). Generally, the younger the Windows version, the more complete are the on-board tools. Additionally, many independent sites such as [download.com](http://download.com), [simtel.net](http://simtel.net), and Microsoft's own [Sysinternals](http://sysinternals.com) provide quite useful command-line utilities, as far as they are not already provided by Cygwin. A few Windows tools, such as **find.exe**, **link.exe** and **sort.exe**, may conflict with the Cygwin versions make sure that you use the full path (`/usr/bin/find`) or that your Cygwin `bin` directory comes first in your `PATH`.

### 3.8.1 Pathnames

Windows programs do not understand POSIX pathnames, so any arguments that reference the filesystem must be in Windows (or DOS) format or translated. Cygwin provides the **cygpath** utility for converting between Windows and POSIX paths. A complete description of its options and examples of its usage are in [?], including a shell script for starting Windows Explorer in any directory. The same format works for most Windows programs, for example

```
notepad.exe "$(cygpath -aw "Desktop/Phone Numbers.txt")"
```

A few programs require a Windows-style, semicolon-delimited path list, which **cygpath** can translate from a POSIX path with the `-p` option. For example, a Java compilation from **bash** might look like this:

```
javac -cp "$(cygpath -pw "$CLASSPATH")" hello.java
```

Since using quoting and subshells is somewhat awkward, it is often preferable to use **cygpath** in shell scripts.

### 3.8.2 Cygwin and Windows Networking

Many popular Cygwin packages, such as **ncftp**, **lynx**, and **wget**, require a network connection. Since Cygwin relies on Windows for connectivity, if one of these tools is not working as expected you may need to troubleshoot using Windows tools. The first test is to see if you can reach the URL's host with **ping.exe**, one of the few utilities included with every Windows version since Windows 95. If you chose to install the **inetutils** package, you may have both Windows and Cygwin versions of utilities such as **ftp** and **telnet**. If you are having problems using one of these programs, see if the alternate one works as expected.

There are a variety of other programs available for specific situations. If your system does not have an always-on network connection, you may be interested in **rasdial.exe** for automating dialup connections. Users who frequently change their network configuration can script these changes with **netsh.exe** (Windows 2000 and later). For proxy users, the open source **NTLM Authorization Proxy Server** or the no-charge **Hummingbird SOCKS Proxy** may allow you to use Cygwin network programs in your environment.

### 3.8.3 The cygutils package

The optional **cygutils** package contains miscellaneous tools that are small enough to not require their own package. It is not included in a default Cygwin install; select it from the Utils category in **setup.exe**. Several of the **cygutils** tools are useful for interacting with Windows.

One of the hassles of Unix-Windows interoperability is the different line endings on text files. As mentioned in Section 3.2, Unix tools such as **tr** can convert between CRLF and LF endings, but **cygutils** provides several dedicated programs: **conv**, **d2u**, **dos2unix**, **u2d**, and **unix2dos**. Use the `--help` switch for usage information.

### 3.8.4 Creating shortcuts with cygutils

Another problem area is between Unix-style links, which link one file to another, and Microsoft .lnk files, which provide a shortcut to a file. They seem similar at first glance but, in reality, are fairly different. By default, Cygwin does not create symlinks as .lnk files, but there's an option to do that, see Section 3.5. These symlink .lnk files are compatible with Windows-created .lnk files, but they are still different. They do not include much of the information that is available in a standard Microsoft shortcut, such as the working directory, an icon, etc. The `cygutils` package includes a **mkshortcut** utility for creating standard native Microsoft .lnk files.

But here's the problem. If Cygwin handled these native shortcuts like any other symlink, you could not archive Microsoft .lnk files into **tar** archives and keep all the information in them. After unpacking, these shortcuts would have lost all the extra information and would be no different than standard Cygwin symlinks. Therefore these two types of links are treated differently. Unfortunately, this means that the usual Unix way of creating and using symlinks does not work with native Windows shortcuts.

### 3.8.5 Printing with cygutils

There are several options for printing from Cygwin, including the **lpr** found in `cygutils` (not to be confused with the native Windows **lpr.exe**). The easiest way to use `cygutils`' **lpr** is to specify a default device name in the `PRINTER` environment variable. You may also specify a device on the command line with the `-d` or `-P` options, which will override the environment variable setting.

A device name may be a UNC path (`\\server_name\printer_name`), a reserved DOS device name (`prn`, `lpt1`), or a local port name that is mapped to a printer share. Note that forward slashes may be used in a UNC path (`//server_name/-printer_name`), which is helpful when using **lpr** from a shell that uses the backslash as an escape character.

**lpr** sends raw data to the printer; no formatting is done. Many, but not all, printers accept plain text as input. If your printer supports PostScript, packages such as `a2ps` and `enscript` can prepare text files for printing. The `ghostscript` package also provides some translation from PostScript to various native printer languages. Additionally, a native Windows application for printing PostScript, **gsprint**, is available from the [Ghostscript website](#).

## 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

---

```
bash$ gcc hello.c -o hello.exe
bash$ hello.exe
Hello, World

bash$
```

---

#### 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)
```

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.

The following is a simple GUI-mode "Hello, World!" program to help get you started:

```
/*-----*/
/* hellogui.c - gui hello world */
/* build: gcc -mwindows hellogui.c -o hellogui.exe */
/*-----*/
#include <windows.h>

char glpszText[1024];

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int APIENTRY WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
    sprintf(glpszText,
        "Hello World\nGetCommandLine(): [%s]\n"
        "WinMain lpCmdLine: [%s]\n",
        lpCmdLine, GetCommandLine());

    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(wcex);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName = NULL;
    wcex.lpszClassName = "HELLO";
    wcex.hIconSm = NULL;

    if (!RegisterClassEx(&wcex))
        return FALSE;

    HWND hWnd;
    hWnd = CreateWindow("HELLO", "Hello", WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL ←
    );

    if (!hWnd)
        return FALSE;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

```
}

return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps);
            RECT rt;
            GetClientRect(hWnd, &rt);
            DrawText(hdc, glpszText, strlen(glpszText), &rt, DT_TOP | DT_LEFT);
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

## 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`

---

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

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 `[?]` 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

---

```
bash$ myprog -t foo --queue 47

bash$ 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

This page gives only a few simple examples of gcc's DLL-building capabilities. To begin an exploration of the many additional options, see the gcc documentation and website, currently at <http://gcc.gnu.org/>

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`).

Fortunately, with the latest gcc and binutils the process for building a dll is now pretty simple. Say you want to build this minimal function in `mydll.c`:

```
#include <stdio.h>

int
```

---



```
hello()
{
    printf ("Hello World!\n");
}
```

First compile mydll.c to object code:

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

Then, tell gcc that it is building a shared library:

```
gcc -shared -o mydll.dll mydll.o
```

That's it! To finish up the example, you can now link to the dll with a simple program:

```
int
main ()
{
    hello ();
}
```

Then link to your dll with a command like:

```
gcc -o myprog myprog.c -L./ -lmydll
```

However, if you are building a dll as an export library, you will probably want to use the complete syntax:

```
gcc -shared -o cyg${module}.dll \
    -Wl,--out-implib=lib${module}.dll.a \
    -Wl,--export-all-symbols \
    -Wl,--enable-auto-import \
    -Wl,--whole-archive ${old_libs} \
    -Wl,--no-whole-archive ${dependency_libs}
```

The name of your library is `${module}`, prefixed with `cyg` for the DLL and `lib` for the import library. Cygwin DLLs use the `cyg` prefix to differentiate them from native-Windows MinGW DLLs, see [the MinGW website](#) for more details. `${old_libs}` are all your object files, bundled together in static libs or single object files and the `${dependency_libs}` are import libs you need to link against, e.g. `'-lpng -lz -L/usr/local/special -lmyspeciallib'`.

### 4.3.2 Linking Against DLLs

If you have an existing DLL already, you need to build a Cygwin-compatible import library. If you have the source to compile the DLL, see Section 4.3.1 for details on having `gcc` build one for you. If you do not have the source or a supplied working import library, 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 controls END
id DIALOGEX memflags exstyle x,y,width,height styles BEG controls END
id DIALOGEX memflags 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 optional 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 MESSAGE TABLE memflags "filename"
memflags defaults to MOVEABLE

```

```
id RCDATA suboptions BEG (string|number) [, (string|number)] (etc) END

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

(User data)
id id suboptions 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 MOVEABLE/FIXED PURE/IMPURE PRELOAD/LOADONCALL DISCARDABLE
```