UnixWare to Linux Porting Guide

Ulrich Drepper
Red Hat, Inc.

1325 Chesapeake Terrace
Sunnyvale
California
94089
drepper@redhat.com

UnixWare to Linux Porting Guide
by Ulrich Drepper

Copyright© 2001 Red Hat, Inc. All rights reserved.

The information contained in this document is provided for informational purposes only.

DISCLAIMER. NEITHER RED HAT OR OTHER PARTIES MAKE ANY REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF RED HAT OR THIRD PARTIES. ANY WARRANTIES WHICH MAY PERTAIN TO SUCH PRODUCTS ARE PROVIDED ONLY UPON THE PURCHASE OR LICENSE OF SUCH PRODUCTS, AND NO WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, AND NON-INFRINGEMENT, ARE EXPRESSLY DISCLAIMED. FURTHERMORE, RED HAT EXPRESSLY DISCLAIM ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. RED HAT DOES NOT WARRANT THAT THIS DOCUMENT IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THRID PARTY PROPRIETARY RIGHTS, AND RED HAT DISCLAIMS ALL LIABILITIES THEREFOR .

RED HAT DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, NOR THAT THIS DOCUMENT WILL BE UPDATED OR MAINTAINED, AND HEREBY DISCLAIM ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Red Hat and other parties may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppel or otherwise, to any intellectual property rights Red Hat or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Red Hat and other parties where appropriate.

Limited License Grant. Red Hat hereby grants you a limited copyright license to download and copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL RED HAT AND OTHER PARTIES HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER RED HAT HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Red Hat Linux is a trademark of Red Hat, Inc.

Unixware is a trademark of Santa Cruz Operation, Inc.

Linux is a trademark of Linus Torvalds.

All other trademarks are the property of their respective owners.

Revision History

Revision 1.0 , 2001-2-25
First published version.

Table of Contents

Chapter 1. Introduction

With the emergence of Linux as a viable computing platform, applications written for other Unix platforms are being ported to Linux. This guide will help you do this task. The focus is on porting applications from UnixWare. SCO's other OS product, Openserver, is not directly corverd in this document but since it is quite similar to UnixWare the information given here should be useful as well. UnixWare is available for the x86 architecture and so is Linux (among other architectures). We will therefore concentrate on this architecture only. This is not really a restriction since except if architecture-specific features are used the ported code will run on all architectures Linux supports.

Generally speaking, porting is quite simple. Because UnixWare is a certified Unix implementation, it has passed the conformance tests of the Unix copyright holders (which SCO still is). Linux is also designed with conformance to the Unix standard in mind. Although Linux has not undergone the conformance testing (due to the costs involved), chances are that if the programmer used only the set of interfaces covered by the Unix standard, you can reuse the code without any changes.

The porting problems that you can expect occur in several different areas:

- The tools used on the different platforms are different. As SCO does not develop tools for Linux, programmers who are using SCO's tools on UnixWare probably have to switch their tools when doing the work on Linux. It might be possible to use the SCO tools to target Linux (since both platforms use the same binary format) but this might proove to be difficult. The problems one can experience are not only related to the actual use of the tools but also to the language the tools accept. SCO might have add extensions to the standard behavior which are not available on Linux.
- The programming interfaces differ. While both operating systems are designed to follow standards, differences in the implementations and different states of the implementations are unavoidable. The programming environment is regulated by a common standard (POSIX.2), but there is still room for differences and extensions.

About this Guide

The following discussion is based on the 7.x series of the Red Hat Linux distribution. This series features the 2.4 version of the Linux kernel and the 2.2 version of the GNU C library. Comparisons with older versions of either package are not discussed in this paper. On the UnixWare side, it is sufficient to discuss the latest version of the OS, UnixWare 7.

Where useful, we discuss upcoming developments on the Linux side. When deficiencies in the Linux system are mentioned, take into account that the development of Linux proceeds very rapidly and the described deficiencies might already be solved. If a particular problem has not yet been solved, this need not prevent you from continuing. Because all of the core operating system is available under an Open Source license, you can either make appropriate changes yourself or contract out the changes. Red Hat's custom engineering services are available for such projects.

Chapter 2. Development Tools

The most notable difference for programmers going from UnixWare to Linux is the change in the development tools. SCO has developed there own compilers and they are available in the OS installation. A C++ compiler is not available in the default installation. The C++ compiler SCO sells is not very standard compliant and no standard feature it supports should be unavailable in the C++ compiler for Linux (GNU C++). To the best of our knowledge does SCO not provide compilers for other languages (like FORTRAN or Java).

Linux developers predominantly use the GNU compiler collection (GCC). The difference is not necessarily a problem, as the quality of the development tools on the Linux side is equally high. It is only the use of SCO-specific features which is truly a problem.

Language Support

Every program that uses only portable language features should not have problems.

C

There should not be any problems at all with compatibility of C programs. The Linux C development and runtime environment is compatible with all the latest standards.

C++

As all compilers are catching up with the latest language standards, incompatibilities due to compilers being at different stages of the race are unavoidable. The compilers available on Linux are very well positioned in the conformance race and changes happen daily. If you require the latest C++ compiler that supports the most language features, it might be worthwhile getting a support contract from Red Hat[1] for the compiler tools.

FORTRAN

There is no FORTRAN compiler available from SCO. There might be third party versions available but we have no access to any such compiler. On Linux the GNU FORTRAN compiler comes with the standard Red Hat Linux distribution, but it currently supports only FORTRAN77. The more recent variants (FORTRAN90, FORTRAN95) are not supported at all. There are commercial FORTRAN compilers available for Linux, but you will need to purchase one.

Java

On Linux the Java support on Linux is very good. There are several JDK implementations available for Linux and, with those, you can execute Java bytecode binaries. Even the performance is comparable. In addition to the option of using bytecode, it is also possible to use the GNU Java compiler to generate native code for the IA-32. The resulting executable is many times faster than the interpreted byte code, even if compiled just-in-time for execution.

An Alternative Way

One way to avoid the troubles with the changing development tools is to use the GNU compilers on UnixWare. The GNU compilers are available for UnixWare and can completely replace the compiler SCO provides.

Many companies are taking advantage of this possibility because it allows retargetting the applications even beyond IA-32. The GNU compiler is available for all modern processors that have at least a 32-bit architecture, whether these are for desktop, server, or embedded systems. This also includes support for the different operating systems for those hardware types.

For companies considering porting to Linux, switching to using the GNU compiler on the known UnixWare platform makes the port much easier. Once the application can be generated on UnixWare using the new tools, you can then attempt to compile on Linux. This will be easier because only API issues (not language issues) can impede progress.

C Compiler Features

If applications were never deployed on other platforms, the code will certainly contain some dependency on the platform on which it was developed. In this section we will discuss the features that are related to the compiler. Information about the system libraries will be given in a later section.

For the compiler we have to handle two compatibility issues:

Invoking the compiler

The command line options to select different modes differs significantly. Especially for writing highly optimized code, it is necessary to know some of the options.

Language Features

Both SCO's compiler and gcc have extended the C language. gcc does this far more, but because the direction of porting is from UnixWare to Linux, this is not an issue.

Invoking the Compiler

In general, there is no standard for the form of command line options of compilers (except the very limited `c89` compiler interface). This leads to wide variations among the different compilers to a point where almost no option is the same on all platforms.

SCO's compiler and gcc agree on the form and function for the following options:

Table 2-1. Common C Compiler Options

`-c`	tells the compiler to compile, but not link
`-o` *`FILE`*	specify output file
`-I` *`DIR`*	add include search directory
`-L` *`DIR`*	add library search directory
`-l`*`name`*	search and add a library with name `libname.a` (or `.so`)
`-A`*`name[(token)]`*	define ISO C assertion
`-E`	only preprocessor phase is performed

`-C`	the preprocessor does not discard comments, they are preserved. This option is with modern versions of gcc only interesting if the preprocessed output is used since the preprocessor is integrated in the compiler and the comments don't have to be preserved since there happens no additional parsing.
`-Dname[=val]`	define preprocessor macro
`-H`	The compiler prints the absolute pathnames of all files includes, one per line. The output is written to standard out. In addition to the file names gcc in some situations gcc prints some more information about files which would benefit from multiple include guards.
`-P`	If the `-P` option is used the preprocessor will not generate #line directives. The line numbers in the output file will therefore match the `.i` file and not the input files. gcc recognizes the same option. But it is necessary to pass the `-E` as well to the compiler. `-P` alone will not have the desired effect.
`-S`	the generate code is not assembled and not linked
`-Uname`	undefine preprocessor macro
`-g`	tell compiler to emit debugging information. SCO's compiler is not capable of optimizing and generating debug information at the same time and does not optimize if `-g` and `-O` are given. gcc can generate debug information for optimized code.
`-O`	In both compilers `-O` instructs the compiler to optimize the code. gcc knows different levels of optimizations which can be specified as a numeric argument to the `-O` option (such as `-O2`). gcc also does not have the limitation that `-O` is disabled for debugging (`-g`) or profiling (`-ql` in SCO's compiler) is selected.
`-E`	the compiler performs only the preprocessing and writes the result to standard output or to the file specified with -o
`-p`	this option instructs the compiler to generate additional code for profiling purposes. When executed the application generates some data which can be examined using the `prof` program. This functionality exists also in gcc the only major difference is that the program is called `gprof` and the file format of the profiling data is different. It is generally advised to use the `-profile` option on Linux but `-p` works as well.
`-S`	instructs the compiler to compile, but not assemble, the code
`-w`	inhibits printing warnings

All the other options the compilers understand are either understood by one side and have no equivalent or are named differently. In the remainder of this section we will cover the most important of these options. To ease the transition to Linux, the list is sorted by the names of the options of SCO's compiler. Names of the option the GNU compiler understands are given in parenthesis if their function does not exactly match that of SCO's compiler.

Table 2-2. Differing Option of the C Compilers

SCO Option	gcc Option	Description
`-V`	`(-v)`	The compiler shows version information for each compilation tool. The gcc equivalent shows the commandline for each tool and its version on standard error. Sometimes the tools the compiler invokes are not the last in the chain. gcc, for instance, normally invokes a tool named `collect` instead of invoking the linker directly. To see the invocation of the linker (done by collect), add the `-Wl,-v` to gcc's command line.
`-w`*`phase`*`,`*`list`*	-W	With the `-W` the user can hand additional parameters to the compilation tools used. Both, SCO's compiler and gcc know this option. Different are the *`phase`* names. preprocessor both compilers use `p` compiler/optimizer the SCO compiler differentiates between compiler and different phases of the optimization. It uses the *`phase`* names `0`, `2`, and `b` for the compiler, the optimizer, and the basic block profiler. gcc has no special option to pass parameters to the compiler since the parameters for it do not have to be specially marked assembler both compilers recognize the *`phase`* name `a` for the assembler. linker both compilers recognize the *`phase`* name `l` for the linker.

SCO Option	gcc Option	Description
`-X`*`str`*	`-ansi/-std/-pedantic`	The SCO compiler allows to control the degree of conformance to the ISO C standard with the `-X` option. gcc has the several different options to do similar things. The `-ansi` directs gcc to turn off compiler features which are incompatible with ISO C90. This includes keywords like asm typeof, and unprotected macros like unix. The `-std` option lets the user choose the standard version the compiler should comply to. Check the gcc documentation for more information. The `-pedantic` option can be used to have the compiler more strictly enforce the ISO C rules.
`-YI,`*`dir`*	`(-nostdinc)`	The `-YI` option causes the preprocessor to use a different default directory to search for include files. The default directory is searched last and is normally something like `/usr/include`. There is no exact correspondence with gcc but it is possible to simulate this. The `-nostdinc` option tells gcc to not look in any of the standard directories. With following `-I` parameters is then possible to specify exactly where the compiler will look. There is only one problem: gcc always uses some special compiler-specific include directories. These must not be excluded. It is possible to determine the complete patch for this directory with `gcc -print-file-name=include`
`-YP,`*`dir`*	`(-nostdlib)`	The `-YP` option changes the default search path of the linker. The default directories are those which are searched after all the directories named by the `-L` or similar options. gcc knows no direct exact equivalent. But with the `-nostdlib` option one can disable the search for libraries in the default directories and with explicit `-L` options it is possible to specify the directories one wants to be used.

SCO Option	gcc Option	Description

`-YS,`*`dir`*	`(-B)`	The `-YS` option allows the user to determine which startup files the linker should pick up. gcc allows the user to select the startup files as well but the option to do this influences other things as well. If the `-B` option of gcc is used the compiler driver searches for all compiler-related files (the programs for the individua phases like `cc1` or `as`) and also the startup files with the prefix given as the parameter to `-B`. It is possible to have multiple `-B` which allows to have programs and the startup files in different locations.
		gcc's `-B` option is much more similar to the *`prefix`*`cc` mechanism SCO's compiler provides. gcc does not change the behavior if the program is invoked with a different name (which is good since this means the compiler does not get confused when sym-links are used). For example, the use of
		
		`foocc`
		`-Ya,/some/dir`
		
		can be emulated when using gcc with
		
		`gcc`
		`-B/some/dir/foo`
		
		The assembler is searched in the given directory. The only difference is that all the other programs (compiler, preprocessor) and the startup files are expected to have the `foo` prefix in the filename as well. If no such file exists the other directories named by `-B` parameters is used and finally the default.

`-Yp,`*`dir`*	`(-B)`	The `-Yp` option allows the user to specify a directory where the preprocessor is found. This can be emulated using gcc's `-B` option. See the description of SCO's `-YS` option above for more information.
`-Y0,`*`dir`*	`(-B)`	The `-Y0` option allows the user to specify a directory where the compiler is found. This can be emulated using gcc's `-B` option. See the description of SCO's `-YS` option above for more information.

SCO Option	gcc Option	Description
`-Y2,`*`dir`*	`(-B)`	The `-Y2` option allows the user to specify a directory where the optimizer is found. Since the gcc has no separate optimizer program this option has no equivalent in gcc..
`-Yb,`*`dir`*	`(-B)`	The `-Yb` option allows the user to specify a directory where the basic block profiler is found. Since the gcc has no separate basic block profiler program this option has no equivalent in gcc..
`-Ya,`*`dir`*	`(-B)`	The `-Ya` option allows the user to specify a directory where the assembler is found. This can be emulated using gcc's `-B` option. See the description of SCO's `-YS` option above for more information.
`-Yl,`*`dir`*	`(-B)`	The `-Yl` option allows the user to specify a directory where the linker is found. This can be emulated using gcc's `-B` option. See the description of SCO's `-YS` option above for more information.
`-A -`	(-ansi)	If SCO's compiler sees the parameter `-A -` it disables all predefined names which pollute the namespace. As far as the preprocessor and namespace is concerned, the same can be achieved with gcc's `-ansi` option. The two options are not equivalent since `-ansi` also influences the language the recognized language and generated messages.
`-Bstatic`	`-static`	Tells the compiler to statically link the application. That is, the link editor will look only for files named `lib*.a`.
`-Bdynamic`	(default)	Tells the compiler to link the application dynamically. That is, the compiler will look first for files named `lib*.so` and, if no such file exists, it will look for files named `lib*.a`.
`-dy/-dn`	(implicit)	The compiler generates a dynamic executable. That is, the runtime linker is used to finish generating the executable. On Linux this happens implicitly. If an object is linked against any shared object, it is a dynamic binary. Similarly, of an object is linked without the use of any shared object, the program is linked statically and the runtime linker is not needed.
`-G`	`-shared`	The linker creates a shared object instead of an executable.

SCO Option	gcc Option	Description
`-KPIC`	`-fPIC`	Tells the compiler to generate position-independent code. With gcc the option is `-fPIC`. There is also an equivalent `-fpic` option but for IA-32 this option makes no difference at all.
`-Kthread`	`-pthread`	If the `-Kthread` option is specified SCO's compiler defines some macros indicating the code is used in multi-threaded applications and links the program with the thread library. gcc recognizes the `-pthread` which has the same effect except that the application is linked with `-lpthread`.
`-KthreadT`		The `-KthreadT` option enables a special tracing mode of SCO's thread library. There is no equivalent in gcc.
`-Kdollar`	(default)	The `-Kdollar` option instructs the compiler to allow dollar ($) characters in identifiers. This is the default in gcc. One would have to disable the use of dollar characters by enabling the ISO C compatibility mode.
`-Ki386`	`-mcpu=i386/` `-march=i386`	Tunes for the i386 processor. The equivalent option For gcc is `-mcpu=i386`. The resulting code will run on all x86 processors that are capable of supporting the mode. To compile code that is even better optimized for the given architecture (but probably will not run on older processors), use the `-march=i386` option.
`-Ki486`	`-mcpu=i486/` `-march=i486`	Similar to `-Ki386`, but for the i486 processor. gcc's `-march=i486` can generate code which does not run on i386 processor machines.
`-Kpentium`	`-mcpu=i586/` `-march=i586`	Similar to `-Ki386`, but for the Pentium processor. gcc's `-march=i586` can generate code which does not run on i386 and i486 processor machines. It is also possible to use `-mcpu=pentium` and `-march=pentium`if this is preferred.
`-Kpentium_pro`	`-mcpu=i686/` `-march=i686`	Similar to `-Ki386`, but for PentiumPro processors. gcc's `-march=i686` can generate code which does not run on i386, i486, and i586 processor machines. It is also possible to use `-mcpu=pentiumpro` and `-march=pentiumpro`if this is preferred.

SCO Option	gcc Option	Description
`-Kblended`	`(default)`	The `-Kblended` option causes SCO's compiler to generate code which runs well on all ix86 processors. gcc provides a much finer granularity of tuning. It is possible to tune the code for a given revision of the architecture while keeping the code portable to all revisions. This can be achieved using the `-mcpu` option (see above). By default the compiler tunes for the architecture it was compiled for.
`-Kieee/ -Kno_ieee`	`-mieee-fp/ -mno-ieee-fp/ -ffast-math`	The `-Kieee` option controls whether floating point operations are executed exactly according to IEEE 754 rules or whether the compiler is allowed to optimize more agressively while relaxing some of the rules. gcc has the option `-mieee` to enforce IEEE 754 compliance. If strict compliance is not needed using the `-mno-ieee` allows the compiler to use some of the features of the FPU better. If the `-ffast-math` option is called the set of performed optimizations is even larger and includes replacing some math library function calls with inlined implementations using the FPU.
`-Kalloca/ -Kno_alloca`	`(-fno-builtin)`	The `-Kno_alloca` option can be used to prevent the compiler from implementing the `alloca()` inline. This is the much more efficient way to handle this but it might be useful. gcc has no specific option to disable the use of `alloca()` (for the reason mentioned). Instead `alloca()` is disabled together with all other builtin functions if the `-fno-builtin` option is used. Another case is if the `-ansi` flag is used since otherwise the namespace rules of ISO C are violated.
`-Kno_frame/ -Kfixed_frame/ -Kframe`	`-fomit-frame-pointer/ -momit-leaf-frame-pointer`	These option determine whether the %ebp register is used as the frame pointer and if it is used, how this happens. Disabling the use prevents stack traces from being made correctly. gcc also allows optimizations to use %ebp as a regular register and not as the frame pointer. This mode can be enabled using the `-fomit-frame-pointer` option. If this optimization is wanted but only if it does not affect the generation of stack traces the `-momit-leaf-frame-pointer` options should be used since gcc will use the %ebp register only for leaf functions which means that stack traces can be generated just fine.

SCO Option	gcc Option	Description
-Kno_args_in_regs/ -Kargs_in_regs	(-mregparm)=*name...*	The -Kargs_in_regs option tells SCO's compiler to divert from the normal calling conventions in some cases and call certain functions in a faster, but incompatible way. The function parameters are passed in registers. gcc allows this as well. One could add the -mregparm option to determine the registers used to pass parameters. But unlike SCO's compiler for -Kargs_in_regs the gcc option works on all functions in the compilation unit. This means that the functions from the cmopilation unit which are called (also) from other objects have a different interface. What one should do instead is to use the regparm attribute to mark the functions one wants to have an optimized interface explicitly. This allows to avoid changing the exported interfaces. See the gcc manual for more information.
-Khost/ -Kno_host	-fhosted/ -ffreestanding/ -fno-builtin	The ISO C standard differentiates between hosted and freestanding implementations. In hosted implementations the compiler can make several assumptions about standard functions. They can be inlined, for instance. gcc has the -fhosted and -ffreestanding options to inform the compiler about this but instead they do not control the treatement of standard functions in the compiler. For this gcc provides the options -fno-builtin and -fbuiltin.
-Kinline/ -Kno_inline	-finline/ -finline-functions	The -Kinline option instructs the compiler to inline certain functions. Which functions are actually inlined is up to the compiler to decide. gcc's equivalent option is -finline-functions. A cost function decides whether a function is worth being inlined or not. The cost function can be influenced by the -finline-limit option. The programmer also can make decisions about inlined functions by marking functions with the inline keyword. If the -finline option is given (which is the default when optimizing) the compiler respects the programmer's choice and inlines functions.

SCO Option	gcc Option	Description
`-Kloop_unroll/ -Lno_loop_unroll`	`-funroll-loops`	The `-Kloop_unroll` options tells the compiler that the optimizations performed should include loop unrolling optimizations. The `-funroll-loops` options does the same for gcc. gcc has another option `-funroll-all-loops` which, as the name suggests, tells the compiler to unroll all loops.
`-Kschar/ -Kuchar`	`-fsigned-char/ -funsigned-char`	The use of the `-Kschar` and `-Kuchar` options decide whether the char type is signed or unsigned. The default is signed. The same is true for gcc only that the options are named `-fsigned-char` and `-funsigned-char`.
`-Kudk/ -Kno_udk`		These options provide a possibility to increase compatibility between OpenServer and UnixWare application. It is therefore not needed for gcc.
`-qp`		SCO's compiler treats this option as an alias of `-p`.
`-ql`	`-ax/ -profile-arcs`	The `-ql` option instructs the compiler to generate code to profile on basic block level. With this execution counts for individual source code lines can be determined. gcc equivalent option is `-ax`. An alternative to `-ax` is to use `-profile-arcs`. This avoids intrumenting blocks whose count can be deduced from other information. The resulting application will run faster and needs less space. One other advantage (and difference) between the two compilers is that gcc allows to instrument optimized code.
`-qf`		This option tells the compiler to instrument the generated code for flow-profiling. gcc has no equivalent option.
`-v`	`-Wall`	If the `-v` option is given the compiler performs more test for ISO C compliance. This is similar to what the `lint` tool would do. gcc has `lint` capabilities built in and they are enabled with the `-Wall` option. gcc can warn about several more problems if additional `-W` parameters are given. They are not considered generally usable enough to be part of the default set of warnings enabled with `-Wall`. See the gcc manual to learn about all the support `-W` options.

SCO Option	gcc Option	Description
-Zp*N*		The options -Zp1, -Zp2, and -Zp4 define the layout of structures. Non-bitfield elements of the structures are aligned to a *N* byte boundary for -Z*N* if the element is larger then *N* bytes. gcc has no command line option which lets the user control the layout in this way. But gcc has the aligned attribute which can be added to every structure element and so make it possible to individually define alignment. See the gcc manual for more information on attributes and specifically aligned.

Language Extensions

The SCO compiler as well as gcc extend the C language. In the following we are describing the extensions of the SCO compiler and possible equivalences on the gcc-side.

Most extensions come in the form of #pragmas. gcc generally does not use #pragmas. The reason is that the old, pre-ISO C99, form of pragmas cannot be generated in macros. This changed with the introduction of _Pragma in ISO C99. Anyhow, gcc uses the keyword __attribute__ which can be used to add information to a definition or declaration of an object. The __attribute__ keyword is followed by two opening parenthesis. The reason for this is that it enables you to define a macro

```
#define __attribute__(ignore)
```

which can be used if the compiler does not understand attributes. More on the syntax can be found in the following table and the gcc manual. All the keywords (like __attribute__ and __aligned__) are given in the form with two leading and two trailing underscore characters. They are also available without underscores or with only two leading underscores. But these names potentially conflict with names in the user programs or the system. The safest possible solution is to use the names used here.

One has to be careful not to miss a use of #pragma in a converted program since gcc simply ignores #pragmas it does not know. Only when the -Wall option is used will it warn about ignored #pragmas.

#pragma weak *name*[,*name*]

This #pragma can be used to specify that a symbol is created weak. gcc also recognizes this #pragma. It is nevertheless advised to use the attribute form:

```
int a __attribute__ ((__weak__)) = 1;
```

#pragma int_to_unsigned *fct*

Marks the function which returns an unsigned value as returning an int. There is no gcc equivalent.

#pragma pack(*n*)

This #pragma specifies that the named structure is packed, i.e., laid out without padding. The gcc way of expressing this is:

```
struct foo
{
  ...
} __attribute__ ((__packed__));
```

This form will pack the structure to the most compact form. But gcc enables you to express even more. One can force individual structure members to be packed while other members are aligned in the normal way. The syntax for this is similar to the following:

```
struct foo
{
  char c;
  short int a __attribute__ ((__packed__));
  short int b;
};
```

In this case the member `a` is not aligned but instead follows immediately the member `a` in memory at offset 1. The member `b` is aligned and follows on offset 4.

Another language extension SCO's compiler provides is the possibility of having assembly code inside the C source files. gcc also provides this functionality but the syntax is in most cases different and both variants' implementation have advantages.

The ISO C standard suggests an implementation of a function like language element. The keyword asm is used in the function name position and the parameter is a string constant. The content of the string must be a sequence of valid assembly instructions. Both compilers provide support for this form of asm expressions. But they are not often useful since it is difficult and unportable at best to access any C non-global objects. One could write could write code assuming a certain stack layout but just enabling another optimization or a compiler upgrade can render the code non-functional. This form is not very useful is general.

For more advanced asm statements it has to be possible to access arbitrary C objects. The two compilers implement this in very different ways.

SCO's compiler allows to write so called asm macros. They look like function definitions but are preceded by the keyword asm. The function body is a sequence of patterns and instructions. The patterns are special as they allow the user specify different assembly code sequences depending on the parameters of the macro. SCO's documentation explains this in more detail. We focus here on explaining the how the features of SCO's compiler can be immitated using gcc.

The syntax of the assembler instructions themselves is the same in both environments. SCO's compiler/assembler recognize the instructions in the AT&T form (as opposed to the Intel form). The GNU utilities on Linux can recognize either form with the default being te AT&T form as well. This leaves only the so called storage mode specifications as a compatibility problem. There are seven different forms supported by SCO's compiler.

treg *`parm`*
ureg *`parm`*
reg *`parm`*

These three forms match if the parameter *`parm`* is in a register. The compiler distinguishes between C register variables and compiler-selected temporary registers. Since gcc takes the register keyword only as a hint this difference does not make sense in gcc.

There is no 100% equivalent way to express what this storage mode specification form does. There if no way to query the compiler whether a certain value is in a register. What is possible is to force a value into a register. Consider the following assembler macro which SCO's compiler would accept:

```
asm void iszero(x)
{
% reg x
  orl x, x
  setz result
% mem x
  cmpl $0, x
  setz result
}
```

gcc has no provisions to distinguish between the value being in a register or being a memory operand. Instead the one or the other has to be chosen. In this case we can force the value *x* in a register. The compiler will generate necessary code to ensure this.

```
#define iszero(x) \
do {                                \
  asm ("orl %1, %1\n"               \
       "setz %0"                    \
       : "=m" (result) : "r" (x)); \
} while (0)
```

This is everything but easy to understand since to do it one has to know a bit about the gcc internals. The first part of the asm looks familiar: it is a string containing the actual instructions. The other two parts are separated by colons. The second part following the first colon specifies so called output parameters. In this case it is a dummy value which is not actually used but which indicates that the asm modifies memory. The third part (the input parameters) is the one of interest here. It says that the value of *x* is passed in a register. The "r" string is indicating register, the "m" string in the second part memory (ignore the equal sign here).

I.e., with a "r" constrain (these strings before the values in the input and output parameter lists) it is possible to force a value in a register and so the orl instruction is operating on a register. If the value passed to `iszero()` is in memory gcc automatically generates an appropriate instruction to load the value into a register before the first isntruction of the asm is executed.

mem *parm*

This storage mode specification is similar to reg et.al. but instead of having the value in a register it is in memory. gcc cannot exactly duplicate this functionality. Here as well all we can do is to force the value to be in the place where we expect it and have gcc move it if necessary. Take the following example which can be compiled using SCO's compiler:

```
asm void mode(val)
{
% reg val
  pushl val
  fldcw (%esp)
  addl $4, %esp
% mem val
  fldcw val
}
```

This can be written with gcc using the "m" contrain for the argument:

```
#define mode(val) \
  asm ("fldcw %0" : : "m" (val))
```

This construct ensures that the parameter *val* will be available in memory before the fldcw instruction and gcc will if necessary store the value in some temporary memory to implement this.

con *expr*

The con storage mode specification matches compile time constants. Instead of a simple variable name it is possible to specify a simple expression containing <, <=, >, >=, ==, !=, % (module), and !% (not module). I.e., the following storage mode lines distinguish a zero value from nonzero ones.

```
asm void set(val)
{
% con val==0
  xorl %eax,%eax
  movl %eax, result
% con val!=0
  movl $val, %eax
  movl %eax, result
% reg val
  movl reg, result
}
```

(Not a very useful example but you get the idea.) The same functionality is available in gcc but has to be implemented very differently. The gcc equivalent of the asm macro above would look like this:

```
#define set(val) \
do {                                                    \
  if (__builtin_constant_p (val)) {                     \
    if ((val) == 0)                                     \
      asm ("xorl %eax, %eax; movl %eax, result");       \
    else                                                \
      asm ("movl %0, %eax; movl %eax, result" : : "i" (val));\
  } else                                                \
    asm ("movl %0, result" : : "r" (val));              \
} while (0)
```

The magic to make this work is the built in function `__builtin_constant_p`. It is no real function. Instead the compiler replaces this call at compile time with a simple zero or nonzero value. The value is nonzero if the expression given as the argument to `__builtin_constant_p` is a constant at compile time. Otherwise the value is zero. Now it should be obvious why this builtin is used here. The content of the if branch of the conditional should be obvious as well. Since *val* is a compile-time constant the expression (val) == 0 also can be evaluated at compile time. Therefore the macro expands to exactly one of the three asms.

This leaves only one question open: what happens if the expression passed to `set` has side effects? The answer is simple: `__builtin_constant_p` does not evaluate the object and so the first if would not cause the side effect to happen. The second if would evaluate the expression including side effects but the then-branch is only executed if *val* is a compile time constant and compile time constants cannot have side effects (otherwise would *val* not be a constant). No it is easy to see that an expression with side effects is executed exactly once which means the semantics is the same as if `set` would be a function call.

lab *str*

This is not a real storage mode specification. Instead it can be used to generate a unique label which can we used inside

Linker Invocation

Next to the compiler, the linker is the most important tool. It controls the final form of the program code and can improve the code generation significantly. Both SCO's and the GNU linker accept numerous options. We'll explain in the following list the most important options of SCO's linker which are not available with the same name in the GNU linker and relate them, if possible, to functionality of the GNU linker. It is also advised that you read the documentation for the GNU linker to find out about the functionality which is not available in SCO's linker. Options which are available in both linkers with the same name and functionality are normally not documented.

Table 2-3. Linker Options

SCO `ld` Option	gld Option	Description
`-a`	`-static`	This option enables the default behavior in the static mode. The linker is creating an executable and undefined symbols cause error messages. GNU ld has the option -static which also enables this behavior.
`-b`	(see comment)	If this option is given the linker does not generate special `-fPIC` relocations for accessing for symbols in shared object which would allow the code to be shared. Instead it creates faster, direct references which cause the text section to become non-sharable. There is no option for the GNU linker to achieve this. It can be achieved, though, by not compiling the source code with the option `-fPIC/-fpic`.

SCO `ld` Option	gld Option	Description
`-Bsymbolic`*`[=list|:filename]`*		Using symbols in shared objects normally always result in the runtime linker performing the relocation, even if the shared object contains a definition for this symbol. SCO's linker allows to specify individual symbols or lists of symbols which are not treated this way. Instead the references are satisfied locally without the dynamic linker being involved. This has speed advantages at runtime. GNU ld has not the same syntax but allows the same kind of optimization with slightly different means. The programmer will have to create a version script and pass it with the `-version-script` option to ld. The named symbol, or even all of them when using wildcards, are resolved locally.
`-Bbind_now`	none available	The `-Bbind_now` option instructs the SCO linker to insert the DT_BIND_NOW tag in the dynamic section of the binary created. This causes the runtime linker to disable lazy relocation and instead perform all relocations at startup time. There is no equivalent in the GNU linker. Users can use the LD_BIND_NOW environment variable at runtime, though.
`-Bexport`*`[=list|:filename]`*	`-rdynamic`	The `-Bexport` and `-Bhide` options control the visibility of symbols in dynamically linked executables and shared objects. By default the linker exports all symbols from shared objects and hides all in executables. The `-Bexport` and `-Bhide` options can be used to selective or generally overwrite these rules. GNU ld has the same default behavior. `gcc`'s `-rdynamic` is equivalent to `-Bexport` in that it instructs the linker to export all symbols. This is only useful when generating executables. When using version scripts it is possible to control the hiding and exporting more exactly on individual symbol level. See the GNU `ld` manual for more information.
`-Bhide`*`[=list|:filename]`*		See the description of `-Bexport` above.

SCO `ld` Option	gld Option	Description
`-Bsortbss`		The `-Bsortbss` option is available in SCO's linker with COFF binary files. There is no need for that on Linux.
`-d` *yn*	`-d` *yn*	With this option it can be decided whether the generated binary is linked statically (is `-d n` is used) or if the binary is using dynamic linker (if `-d y` is used). GNU ld understands the same option.
`-e` *symbol*	`-e` *symbol*	This option can be used to determine the entry point in the object. GNU ld has the same option.
`-f udk`	not needed	The SCO linker allows this option to be used to select certain binary compatibility features coming up with their different versions of Unix (UnixWare and OpenServer). None of this is interesting for Linux, of course.
`-f osr5`	not needed	Just like `-f udk` this selects one of SCO's ABI. Not needed here.
`-f iabi5`	not needed	Just like `-f udk` and `-f osr5` this selects one of SCO's ABI. Not needed here.
`-G`	`-shared`	Generate a shared object. The equivalent for the GNU linker is `-shared`.
`-h` *name*	`-h` *name*	This option allows to specify the soname when generating shared objects. GNU ld understands the same option.
`-I` *name*	`--dynamic-linker` *name*	Set name as the interpreter in the program header of an executable. The equivalent option for the GNU linker is `--dynamic-liner` *name*.
`-m`	(`-M`)	Print a linker map. The `-M` option prints something comparable but with a different format and slightly different content.
`-M` *mapfile*	`-T` *mapfile*	The `-M` option allows the user to specify a linker map file which the linker will use to form the output file. The use of this option is discouraged by SCO but if people still want to use it they will find in the `-T` option the GNU equivalent. But the input file format will be different and therefore an easy transition is not possible.
`-Qy`	(ignored)	SCO's linker adds a string identifying the linker to the comment section of the generated binary. This option is ignored for compatibility by the GNU linker. To emulate the behavior one could create such a string in the input files (using the `#ident` directive) or use the linker script to add it automatically.

SCO `ld` Option	gld Option	Description
`-s`	`-S/-s`	This option instructs the linker to strip symbolic information from the output file. The GNU linker has a finer grain for this functionality. If the `-s` option is used, the GNU linker strips all symbols from the file. This is much more than SCO's linker does. To get the equivalent of the `-s` option, one has to use `-S` with the GNU linker, which removes only the debugging information.
`-t`		This option allows you to turn off warnings about multiply defined symbols that are not the same size. There is no equivalent in the GNU linker.
`-x`		The `-x` option instructs SCO's linker to remove all locale symbols and all debugging information. There is no direct equivalent to this in GNU ld but stripping with `-s` goes a long way in the same directoin.
`-z defs`	`--no-undefined`	Undefined symbols at the end of the linking process will cause a fatal error. This option is ignored by the GNU linker and the user has to use the option `--no-undefined` to achieve the same results.
`-z nodefs`		When this option is given SCO's linker allows undefined symbols when building shared objects. There is no equivalent optoin but unless `--no-undefined` is passed to GNU ld this is the default mode.
`-z text`		When linking dynamically this option causes the linker fatally if the resulting binary would contain any relocations against a read-only section. There is no direct equivalent to this in GNU ld but it is easy enough to check for text relocations with `objdump` `objdump -p` *`file`* `| grep TEXTREL`

Notes

1. http://www.redhat.com/services/gnupro/gnupro_plus.html

Chapter 3. System Interfaces

The system interfaces, as defined by the standard libraries such as `libc`, `libm`, and `libpthread`, make Linux appears as an almost complete Unix system. There are only a few function families and individual functions missing and a very small number of functions have a different, possibly limited behavior.

In this section we will first introduce the functions which are entirely missing on Linux. We will only cover the standard interfaces and a few very important and widely used other interfaces.

We will discuss interfaces which are different or have a limited functionality (mostly on the Linux side). Knowing about this can save a lot of debugging time.

But before we start, a few more words on the standards. UnixWare is a certified Unix implementation and Linux is also modeled after Unix. The Unix standard is published by the OpenGroup. This huge document governs almost all interfaces the standard libraries provide. This is at least true on the UnixWare side, Linux has several extension. To enable the Unix interface, the C preprocessor macro _XOPEN_SOURCE must be defined to the value 500. This selects the interface of the fifth revision of the Unix standard. Adventurous people can already use the interface of the sixth revision by setting the symbol _XOPEN_SOURCE to the value 600 (but only on Linux).

The Unix standard is in large parts based on the POSIX standards ISO 9945-1 and 9945-2 (aka IEEE 1003.1 and IEEE 1003.2) with their numerous extensions. This means that Unix systems also implicitly implement POSIX.

Below POSIX, mainly because of history, further interfaces are available. These are for BSD and SVID systems. Today none of these last interfaces should be used directly since they are long obsolete. For programming one should either use the POSIX or the Unix interface.

One last interface is specified and this is the GNU interface. It includes everything Unix does and more. The extra interfaces not included in the Unix specification are not portable and one should know exactly when and why to use them.

Interfaces Missing on Linux

The Linux system interface lacks at the time of this writing some interfaces which are available on UnixWare. These fall into two categories: non-standard interfaces and standard interface.

Into the former category falls one entire special and popular library: UnixWare' non-POSIX thread library. The file in question is `libthread.so` (`libthread.a`) and the system header is <`thread.h`>.

The interface of this thread library can to some extent be mapped to functions in the POSIX thread library (which UnixWare also provides). But there are exceptions. The first table provided here maps functions where there exists a corresponding interface in the POSIX thread library of Linux.

Table 3-1. UnixWare Thread vs. POSIX Thread Library

UnixWare Thread Library	Linux POSIX Thread Library
`thr_create()`	`pthread_create()`
`thr_exit()`	`pthread_exit()`
`thr_getprio()`	`pthread_getschedparam()`

UnixWare Thread Library	Linux POSIX Thread Library
thr_getspecific()	pthread_getspecific()
thr_join()	pthread_join()
thr_keycreate()	pthread_key_create()
thr_keydelete()	pthread_key_delete()
thr_kill()	pthread_kill()
thr_self()	pthread_self()
thr_setprio()	pthread_setschedparam()
thr_setspecific()	pthread_setspecific()
thr_sigsetmask()	pthread_sigmask()
thr_yield()	sched_yield()
mutex_destroy()	pthread_mutex_destroy()
mutex_init()	pthread_mutex_init()
mutex_lock()	pthread_mutex_lock()
mutex_trylock()	pthread_mutex_trylock()
mutex_unlock()	pthread_mutex_unlock()
rmutex_destroy()	pthread_mutex_destroy()
rmutex_init()	pthread_mutex_init()
rmutex_lock()	pthread_mutex_lock()
rmutex_trylock()	pthread_mutex_trylock()
rmutex_unlock()	pthread_mutex_unlock()
rwlock_destroy()	pthread_rwlock_destroy()
rwlock_init()	pthread_rwlock_init()
rwlock_rdlock()	pthread_rwlock_rdlock()
rwlock_rwrlock()	pthread_rwlock_wrlock()
rwlock_tryrdlock()	pthread_rwlock_tryrdlock()
rwlock_trywrlock()	pthread_rwlock_trywrlock()
rwlock_unlock()	pthread_rwlock_unlock()
cond_broadcast()	pthread_cond_broadcast()
cond_destroy()	pthread_cond_destroy()
cond_init()	pthread_cond_init()
cond_signal()	pthread_cond_signal()
cond_timedwait()	pthread_cond_timedwait()
cond_wait()	pthread_cond_wait()
barrier_destroy()	pthread_barrierattr_destroy()

UnixWare Thread Library	Linux POSIX Thread Library
barrier_init()	pthread_barrier_init()
barrier_wait()	pthread_barrier_wait()
_spin_destroy()	pthread_spin_destroy
_spin_init()	pthread_spin_init
_spin_lock()	pthread_spin_lock
_spin_trylock()	pthread_spin_trylock
_spin_unlock()	pthread_spin_unlock

The difference between the mutex_* and rmutex_* function in SCO's library is that the latter are recursive mutexes. In Linux's POSIX thread library this is handled by the Unix extensions of POSIX mutexes which allow setting the type to recursive (using pthread_mutexattr_settype() and PTHREAD_MUTEX_RECURSIVE or through the initializer PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP).

No corresponding interfaces exist for the following functions:

thr_suspend()

The thr_suspend() function is stops the thread specified by a parameter. This is a big problem since the function cannot take into account mutex and other synchronization objects the thread is currently allocating. But this can lead to deadlocks since other, not stopped threads might depend on the synchronization objects.

For this reason the suspend function was not added to the POSIX standard (and it also gets removed from specifications such as the Java thread library).

thr_continue()

This interface is used to continue a stopped thread. Since thr_suspend() is rejected it is unnecessary for the same reasons to define a thr_continue() equivalent.

thr_main()

This interface can be used to determine whether the current thread is the main thread or not. There is no equivalent in the POSIX thread library and also not in Linux's implementation.

thr_min_stack()

There is no corresponding interface in the Linux threads implementation. But the Unix standard requires a symbol PTHREAD_STACK_MIN to be defined.

thr_getconcurrency()
thr_setconcurrency()

These interfaces are only necessary for m-on-n implementation (m user level threads on top of n kernel threads, m >= n). Since the Linux thread library is currently not designed in this way, this interface serves no purpose. This might change in future, though.

`thr_get_rr_interval`

The `thr_get_rr_interval` was not accepted in the POSIX standard. Instead POSIX includes the function `sched_get_rr_interval`. On Linux for the current implementation of the thread library the functionality of this function is equivalent with the `thr_get_rr_interval` function since every thread runs in its own process. This might change in future, though.

`thr_getscheduler`
`thr_setscheduler`

The situation with `thr_getscheduler` and `thr_setscheduler` is similar to that of `thr_get_rr_interval`. The functions were not adopted into POSIX and there are the functions `sched_getscheduler` and `sched_setscheduler` which in the moment are equivalent on Linux.

`_barrier_spin`
`_barrier_spin_destroy`
`_barrier_spin_init`

Spinning barriers as handled by these functions are generally a bad idea. The POSIX standard does not has support for them and therefore the Linux POSIX thread implementation does not have them either. It is much better and portable to use the blocking barrier functions.

The situation with the semaphore interfaces is similarly. SCO has its own special implementation which in parts can be mapped to the POSIX interfaces. Just as for the thread stuff, SCO provides besides the own interface a POSIX semaphore interface.

One major difference between the UnixWare and the Linux interface is that UnixWare defines these functions in `librt` while on Linux they are in `libpthread`.

Table 3-2. UnixWare Semaphore vs. POSIX semaphore Library

UnixWare Semaphore Library	Linux POSIX Semaphore Library
`sema_destroy()`	`sem_destroy()`
`sema_init()`	`sem_init()`
`sema_post()`	`sem_post()`
`sema_trywait()`	`sem_trywait()`
`sema_wait()`	`sem_wait()`

Further differences in the interfaces of the thread library result from missing functionality. This will be covered in the next section.

One other big area where there is, to some extent, no equivalent functionality is STREAMS and TLI (transport layer interface). The STREAMS network interfaces are not available by default on Linux systems. However, the C library provides the interface but they will always fail unless the kernel extension implementing STREAMS is available.

If the program to be ported uses STREAMS, probably the best solution is to rewrite the networking part to use the basic POSIX socket interface. This does not only make the program use a POSIX interface, it also will improve the portability.

If rewriting is not an option or if a short-term solution is needed, information about the Linux STREAMS implementation can be found at the Linux STREAMS[1] site.

The TLI implementation which is part of the Linux STREAMS implementation may or may not be compatible with the SysV specification. Red Hat has no experience

whatsoever with this code and we are not advising to use it.

Differing Interfaces Between UnixWare and Linux

Some of the interfaces available on UnixWare and Linux differ despite having same name and same purpose. This can happen for three reasons:

- limitations of the implementation (mainly on the Linux side)
- different interpretation of the standard
- extension on top of a standard implementation

In the remainder of this section we will outline a few of the functions falling into this category. This list will most probably be incomplete. In most cases if a Linux implementation differs from a UnixWare implementation, the differences were unintentional and will be removed when reported. Therefore if you have found a difference, it was probably not previously known. We will concentrate here mainly of those differences where functionality is missing because the underlying system does not support the operation.

Limited Implementations

No Process-Shared Synchronization Objects

The functionality which will probably be missed most when coming from UnixWare are sharable synchronization objects. This means mutexes, semaphores, and conditional variables which can be shared between different processes (not threads, this of course works).

Calls to `pthread_mutexattr_setpshared`, `pthread_rwlockattr_setpshared`, and `pthread_condattr_setpshared` will fail if the second parameter is set to PTHREAD_PROCESS_SHARED. This functionality is not implemented in the thread library because the kernel implementation is lacking some features. Once this kernel limitation is lifted the functionality is available. Therefore it is not a good idea to completely disable all uses of these functions for Linux. Instead a check of the return value at runtime should be used to determine whether the functionality is available or not.

This same problem exists for standard interfaces which are not yet supported by UnixWare (like `pthread_barrierattr_setpshared()`).

The only exception is the implementation of the spinlocks. The second parameter of `pthread_spin_init` can be set to PTHREAD_PROCESS_SHARED and the function will not fail.

Signal in Threaded Application

The Linux POSIX thread implementation is using individual processes for each thread. These are not completely separated processes but instead since they have to shared things like virtual memory, file descriptors and the like.

However, the kernel does not really know the difference between threads and processes. Therefore it does not handle the delivery signals correctly. There is no single process ID (each thread has its own which is another difference from the POSIX standard) and the kernel is delivering the signal to that thread.

There is currently no easy way out of this. Programs which depend on signal delivery will still work but all the signals are received by exactly the thread with the process ID that was used.

This is the status at the time of this writing. It might be that this problem is already worked around since it is a quite high-priority problem and will be worked on when possible.

Unless absolutely necessary to implement a short-term solution no program should depend on this non-standard behavior. Future implementations which will hopefully be ABI compatible will implement the right behavior and programs assuming the current non-standard behavior will break.

Linux Development Environment Namespace Issues

Great care has been taken to ensure the namespace of the development environment is clean and compliant with the individual standards. This means that only the functions specified by the standards are made available when the appropriate feature-select macro for the standard is selected, and extended interfaces are only enabled when explicitly requested.

The available and useful feature select macros are those in the following table. They have to be defined (as C preprocessor macros) before including the first system header. The best way to do this is to add, e.g., -D_GNU_SOURCE to the commandline of the compiler.

`_ISOC99_SOURCE`

This macro selects makes all the functionality of the ISO C99 standard available.

> **Note:** The GNU C library includes all ISO C99 functionality. This introduces two kinds of problems: a) traditionally used interfaces might be reused (example: the `nan` function, there was a `nan` symbol on some implementation denoting the NaN value); b) silent changes in the implementation. The latter is especially bad but unavoidable. One often hit problem is the change in `strtod` et.al to allow the hexadecimal floating-point number notation which suddenly lets `strtod` accept expressions it would not have before.

`_POSIX_SOURCE`

Signals that POSIX.1 functions should be used. This is not very useful and `_POSIX_C_SOURCE` should be used instead.

`_POSIX_C_SOURCE`

This macro should be set to a value representing the date of the revision of the POSIX version one wants to use. Currently the last officially supported version is identified by the value 199506L. To select this revision one should add `-D_POSIX_C_SOURCE=199506L` to the command line.

There are more revisions of the standard coming along and it will become necessary to set the macro to higher values. But the work of the standards committee is not yet finished. See the description of `_XOPEN_SOURCE` for more information.

`_XOPEN_SOURCE`

This macro can be used to select between the interfaces of the various revisions of the X/Open Portability Guides (XPG) and the Single Unix Specification. The GNU C library implements only the XPG4 and Single Unix Specification interfaces.

Normally everybody wants to set this macro to the value 500 which selects the Single Unix interface. But it is also possible to set this macro to 600. This will select the interfaces for the next revision of this specification. This is especially interesting since the next revision of this specification will be unified with the POSIX standard. I.e., the value 600 will select also all the new POSIX interfaces. Some of these new interfaces are critical for high-performance applications and therefore this option is useful even though the specification is not yet finished.

`_LARGEFILE_SOURCE64`
`_FILE_OFFSET_BITS=`*m*

These macros enable the interfaces agreed on by all Unix vendors at the Large File Summit. They exist to enable 32-bit systems, which traditionally use types which limit them to files of sizes up to 2GB, to support large files. There are two modes in which this can happen.

The first mode it to define `_LARGEFILE_SOURCE64` which makes an additional set of functions and data types available. The new data types are ino64_t, off64_t, blkcnt64_t, fsblkcnt64_t, fsfilcnt64_t, and all composed types which contain at least one element with of the basic forms of the former types (e.g., struct stat contains an element of type off_t and therefore exists a type struct stat64).

The naming of the functions to use with these types follows the same scheme. There exists a `lseek64()` function which takes and returned off64_t values. There also is a `open64` functions which opens a file ready for these large file operations.

The second mode does not require cleanly written applications to be rewritten. Define `-D_FILE_OFFSET_BITS=64` on the compiler command line enables a mode in which the `*64()` functions and appropriate types completely replace the old interfaces. I.e., off_t is suddenly a 64-bit type and `lseek()` takes and returns such a value.

For more information about this extension, read the Large File Summit document which is available, for instance, as part of the Single Unix Specification.

`_GNU_SOURCE`

If this macro is defined, all of the previously mentioned interface plus several GNU-specific are made available. This provides the most convenient programming environment but programs are becoming less portable. Therefore care should be taken when writing applications which are expected to run on other systems as well.

To ensure programs are not accidentally using interfaces which they should not use, it is important to select the appropriate feature select macro and then use the `-Wall` option to catch all occurrences of functions without prototypes. All such cases (unless they are error in the user code) point to possible problems since a symbol or function outside the selected interface is used.

At link time the GNU C library already tries to make sure that internal interfaces cannot be used. This is done by not exporting these interfaces and therefore not providing the linker to bind the application to these symbols. There are two problems:

1. Not all interfaces can be hidden. Some interfaces are needed by other shared objects implementing the system libraries (such as the thread library). This does not mean that these interfaces can be used. Since all system libraries are released in synch this means that the internal, but exported, interfaces can change since the necessary changes can be performed in all system libraries.

   The general rule is that user programs must not use any of the interfaces whose name is in the system namespace. This means, all symbols with a leading underscore character must be used. There are only a very few exceptions:

   `_tolower()`
   `_toupper()`

   Traditional Unix interface that, on older systems, is faster than `tolower()` and `toupper()` resp. There is no reason to use this interface on modern systems.

   `_exit()`
   `_Exit()`

   The variation of the `exit()` interface is needed in some special situations. The second form is the one chosen for the same functionality by the ISO committee.

   `_setjmp()`
   `_longjmp()`

   These are the old 4.3BSD compatible names for the modern `sigsetjmp()` and `siglongjmp()` interfaces where the second parameter is nonzero.

   No other symbol starting with an underscore character should be used directly by the application.

2. Linking statically will not show the same behavior since the symbol export restrictions only work with shared objects. Every single symbol of the `libc` is available when linking statically. This is not a problem since all dependencies are added to the application (and therefore changes in the `libc` implementation are not effecting the application) but one of course has all the disadvantages of static linking.

Notes

1. http://www.gcom.com/LiS/