

Porting Open64 to the Cygwin Environment

Nathan Tallent Mike Fagan *

November 2003

Abstract

Cygwin is a Linux-like environment for Windows. It is sufficiently complete and stable that porting even very large codes to the environment is relatively straightforward. Cygwin is easy enough to download and install that it provides a convenient platform for traveling with code and giving conference demonstrations (via laptop) – even potentially expanding a code’s audience and user base (via desktop). However, for various reasons, Cygwin is not 100% compatible with Linux. We describe the major problems we encountered porting Rice University’s Open64/SL code base to Cygwin and present our solutions.

1 Introduction

Cygwin is a Linux-like environment for Windows. It provides a Windows DLL that emulates nearly all of the Linux API. Programs written for Linux can link with the Cygwin DLL and thus run on Windows. A reasonably proficient Unix user can easily download and install a relatively complete Linux environment, including shells, development tools (GCC, make, gdb), editors (emacs, vi) and a graphical environment (XFree86). Moreover, the environment is complete and stable enough that it is relatively straightforward to port even very large codes to Cygwin. Because of the prevalence of Windows-based desktops and laptops, Cygwin provides an attractive means for traveling with code and demonstrating it at conferences – even potentially expanding a code’s audience and user base. However, for full-time development or for critical applications, Unix remains superior. More information about Cygwin can be found on its web site, <http://www.cygwin.com>.

Because Cygwin is an *emulator*, it implements the Linux API through Windows system calls. In other words, Cygwin retains all the subtle incompatibilities, restrictions, and limitations that ‘Windows’ implies vis-a-vis ‘Linux’. Windows is always the underlying OS and, not surprisingly, Windows runs Windows programs better than Cygwin emulates Linux programs. However, when used for the applications described above, Cygwin’s advantages far outweigh its disadvantages.

This report summarizes our experience porting Rice University’s Open64/SL code base to Cygwin. More properly, it summarizes our experiences porting the Fortran 90 front-end (mfef90), the WHIRL tools (ir_b2a, ir_size) and the whirl2f back-end.

*Thanks to LACSI and ACTS for funding this work

The Open64 code base originated with Silicon Graphics and Cray and was eventually released to the public under GNU's General Public License. The code was written primarily in C and legacy C++ (pre-ANSI/ISO C++ standard) and targeted especially to the IRIX and UNICOS environments, though the code had been quickly 'ported' to Linux.

Already, the observant reader will have identified some of the major contours of the Cygwin-Open64 terrain. We have grouped the issues under four sections which are discussed in detail throughout the rest of the report.

- Problems Related to Differences and Limitations of Cygwin vis-a-vis Linux.
- Problems Related an Incomplete Cygwin API
- Problems Related to Non-standard C/C++ and Specialized Unix System Calls.
- Other Problems

2 Problems Related to Differences and Limitations of Cygwin vis-a-vis Linux.

2.1 Introduction

While it is not our intention to provide a general introduction to Cygwin programming, it will be helpful to quickly detail a few important differences between Cygwin and Linux. For more information, refer to the Cygwin documentation at <http://www.cygwin.com>

- System macros
 - Cygwin compilers set the `__CYGWIN__` macro.
- File system
 - DOS (and Windows) reserves certain file names such as `AUX`, `COM1`, `LPT1` or `PRN` for devices. These file names are also invalid under Cygwin.
 - As in Windows, binaries automatically have the `.exe` extension. Unless for some reason there is ambiguity – i.e. both `moo.exe` *and* `moo` exist – the two names are interchangeable for the purpose of launching programs. Moreover, GCC will automatically add the extension (if needed). However, the two files may be considered distinct for other purposes, such as deleting a file.
 - Windows and Unix paths are radically different, with Windows tied to DOS's concept of drive letters. Cygwin is generally able to transparently convert between Windows paths (e.g. `Z:\moo`) and a simulation of POSIX paths (e.g. `/cygdrive/Z/moo`). The `cygpath` utility can help in other cases.

- Windows ‘shortcuts’ do not translate into Unix symbolic links. Cygwin supports symbolic links by creating files with a special magic cookie; these links will not work under Windows. The NTFS file system fully supports hard links, but on FAT, the file is actually *copied*.
- DLLs
 - Cygwin uses DLLs not DSOs. The `.dll` extension is *important*.
 - As on Linux, the `LD_LIBRARY_PATH` environment variable is used by `dlopen()` to find DLLs, but unlike Linux it is *not* used by the runtime loader. The `PATH` environment variable is used for the latter purpose.
 - The `cygcheck` utility simulates the functionality provided by the `ldd` command on Unix, indicating whether or not all a program’s DLL dependencies can be resolved.

2.2 EXEs, DSOs and DLLs

We had to change the Open64 build files to consciously give executables the `.exe` extension and to create DLLs instead of DSOs. Although GCC automatically adds the `.exe` extension if necessary, the complete name of the binary must be known if it is to be deleted when removing build files with `make clobber`. Moreover, on Cygwin, one creates DLLs, not DSOs. The procedure for creating DLLs is the same as creating DSOs on other modern Unices, but the extension is crucial. (See section 6 for a sample makefile that creates DLLs using GCC.) We initially encountered all sorts of strange linking errors, but changing the extension from `.so` to `.dll` resolved nearly all of them.

Although Cygwin is very similar to ia32-Linux, these previously mentioned facts placed a wedge between Cygwin builds and the ia32-Linux builds. Because Open64 was never originally designed to be particularly portable, it could store build-specific configuration information in completely separate build trees without great inconvenience. This meant that each supported platform had its own *manually configured* build tree such as

```
Open64/osprey1.0/targ_mips_irix
Open64/osprey1.0/targ_ia32_ia64_linux
Open64/osprey1.0/targ_sparc_solaris
```

Consequently, it seemed that we would need to create an entirely new build tree for ia32-Cygwin. Because 1) creating whole new build trees is both cumbersome and a maintenance headache and 2) we did not anticipate anyone wanting or trying to build Open64 on Linux and Cygwin *within the same file system*, we decided to use the ia32-Linux build directory but add some platform detection code within

```
Open64/osprey1.0/Makefile.gsetup
```

This file, included by other makefiles, checks for the Cygwin platform and sets variables representing what extensions binaries and DSOs ought to have. These variables are then used in the platform-independent makefiles located within the source tree.

Because Open64's back-end loads DSOs (or DLLs) using `dlopen()`, we also had to change the code to use the appropriate extension, conditional upon the build environment.

2.3 Linking and Weak Symbols

Open64 was not designed to build 'self-contained' DSOs, i.e., DSOs that are not linked with other Open64 DSOs. For example, several back-end source files refer to routines defined in many other parts of the compiler. Needed symbols remain undefined until an appropriate call to `dlopen()` is made. For example, when running `whirl2f`, the back-end driver determines that it needs to load the `whirl2f` DSO, which resolves `whirl2f` symbols and allows the translation to proceed. Back-end symbols not needed for `whirl2f` may remain undefined.

The 'problem' with this scheme is that it requires either that the binary format supports tagging these undefined symbols as weak or that the linker allows user symbols in a DSO to remain unresolved. Neither of these is an option under Cygwin: Windows PE-COFF binaries do not support weak symbols and, unlike Linux, the Cygwin linker will complain when DSOs contain unresolved symbols. Because of this, to accomplish the Cygwin link without massive code rewriting, we extended a weak symbol workaround-hack that was already in place for versions of GCC that did not support weak symbols. More specifically, macros of the form

```
#if defined(_GCC_NO_PRAGMAWEAK)
```

were extended with

```
#if defined(_GCC_NO_PRAGMAWEAK) || defined(__CYGWIN__)
```

The actual amount of code needed to resolve this issue was relatively small, but because the weak symbol workaround code is scattered throughout the Open64's back-end – and beyond! – and because it can be difficult to understand within Open64, it is worthwhile to consider a small example.

We simulate weak function symbols by creating function pointers for each symbol that should be weak and then renaming all *uses* of the symbol to refer to the new function pointers. When the definitions of these function pointers are included in the DSO needing weak symbols, the function pointers will be available and the link will not fail. It should be noted, though, that these new function pointers currently point nowhere.

To illustrate, assume we have two DSOs, `DSO1` and `DSO2`, each implemented with one source file that contains one function. Here is the source code for each file (`ds01.h` and `ds02.h` contain straightforward declarations of their respective functions.):

```

// *** dsol.C ***
#include <iostream>
#include "dsol.h"

void dsol()
{
    std::cout << "<dsol() says hello!>\n";
}

// *** dso2.C ***
#include <iostream>
#include "dsol_weak_hack.h" /* instead of dsol.h */
#include "dso2.h"

void dso2()
{
    std::cout << "[dso2() makes an appearance]\n";
    std::cout << " [dso2() calls dsol()...] ";
    dsol();
}

```

Clearly DSO1 is independent and DSO2 is dependent on DSO1. However to simulate Open64, we cannot link DSO2 against DSO1. In order to retain DSO2's independence, we simulate weak symbols by replacing the use of `dsol` with a function pointer whose definition is available. DSO2 can then be linked using this alternative definition. The following code implements this scheme.

```

// *** dsol_weak_hack.h ***
// 'Weak' symbols from <dsol.h>
extern void (*dsol_p)(void);

// Replace original symbols with our 'weak' symbols
# if !defined(DISABLE_DSOL_SYMBOL_RENAMING)
# define dsol (*dsol_p)
# endif

// *** dsol_weak_hack.C ***
// Define the bogus function pointer
void (*dsol_p)(void);

```

Thus far we have created two independent DSOs. DSO1 is simply created by compiling one source file; DSO2 is created by compiling two source files – `dso2.C` and `dsol_weak_hack.C` – and linking them together.

However, for the scheme to work, any executable using DSO2 must initialize the ‘weak symbols’ to actually point to the real symbols. (Thus far, any use of our alternative function pointers will cause some sort of runtime trap!) This can be accomplished using a static initializer to set all the ‘weak symbols’ – which are masquerading as function pointers with different names – to point to the real symbols.

```
// *** test_weak_hack_init.C ***
#define DISABLE_DSOL_SYMBOL_RENAMING 1
#include "dsol_weak_hack.h"
#include "dsol.h"

// Initialize the function pointers simulating weak symbols
struct DSOL_INIT {
    DSOL_INIT() { dsol_p = dsol; }
};

DSOL_INIT dsol_initializer; // the static initializer
```

The above initializer, along with the two independent DSOs already created, can be linked with the following program to produce the output given below.

```
// *** test.C ***
#include <iostream>
#include "dsol.h"
#include "dso2.h"

int main()
{
    std::cout << "main() calls dsol() and dso2()...\n";
    dsol();
    dso2();
    return 0;
}

// *** Output ***
[nesini]:(dsotest1)> ./test.exe
main() calls dsol() and dso2()...
<dsol() says hello!>
[dso2() makes an appearance]
[dso2() calls dsol()...] <dsol() says hello!>
```

For a sample GNU Make makefile that can be used with the above code, see section 6.

2.4 Very Strange Linking Errors

When linking the Fortran 90 front-end, mfef90, we initially encountered some *cryptic* linking errors.

```
fu000001.o(.idata$3+0xc): undefined reference to `__libc_iname'
fu000002.o(.idata$3+0xc): undefined reference to `__libc_iname'
fu000003.o(.idata$3+0xc): undefined reference to `__libc_iname'
fu000005.o(.idata$3+0xc): undefined reference to `__libc_iname'
fu000007.o(.idata$3+0xc): undefined reference to `__libc_iname'
fu000008.o(.idata$3+0xc): more undefined references to `__libc_iname' follow
```

It turns out that Cygwin's version of GCC generates these errors when a standard external symbol that is to be located in the data section is manually declared instead of with the appropriate header file. More precisely, it seems (for a reason we did not take the time to fully investigate) that in order to link properly with system DLLs, such symbols need to be marked with a 'dllexport' tag. One can do this manually, or let the headers do it.

In our case, the specific problem was that the external symbols related to `getopt()` – `optarg`, `optind`, `opterr`, `optopt` from `<unistd.h>` – were declared *manually*:

```
extern int optind;
/* ... */
```

Because it is already difficult to justify not using the standard header, we simply replaced the manual declarations with the appropriate header and were able to link.

2.5 API Differences

Because Cygwin is implemented by the underlying Windows API, there are some semantic differences between certain Linux and Cygwin system calls. We encountered one instance where this proved significant.

In order to write WHIRL files to disk, Open64 uses a memory map. It originally took advantage of an SGI extension to `mmap()` that would automatically increase the size of the memory-mapped file. In order to support standard implementations of `mmap()`, the code was extended to use `ftruncate()` both to preallocate a chunk of disk space, and to grow the file if necessary. In both cases, after all data was written, the file was truncated to the exact size using `ftruncate()`.

There were two problems with the code. First, it was so badly hacked that there were *three* versions of code for only *two* distinct needs, an SGI version and a non-SGI version. We ripped out one version and cleaned up the macros. The second problem was that, because the Windows API restricts the operations that can be performed on an open file, `ftruncate()` failed every time it was used on a memory-mapped file. The solution was simply to unmap the file using `munmap()` before truncating the file to its new size.

```
Open64/osprey1.0/common/com/ir_bwrite.cxx
Open64/osprey1.0/common/com/ir_bcom.cxx
```

3 Problems Related an Incomplete Cygwin API

Although fairly complete, Cygwin does not yet support the whole Linux API.

3.1 Message Catalog System

Cygwin does not support the message catalog system provided by the user command `gencat` and the system calls `catopen()`, `catclose()` and `catgets()` from `<nl_types.h>`. One of the advantages to using this functionality is that it is designed to automatically adjust message reporting based on locale. For example, if Open64's `mfef90` had both both English and German versions of its error messages, the the message retrieval function would automatically select the correct version based on settings in the user's environment. Another advantage is that users can store messages in a relatively maintainable format, using system commands to generate the actual data used by message retrieval calls. The original message data file is

```
Open64/osprey1.0/crayf90/fe90/cf90.msgs
```

Because `mfef90` originally used the SGI-specific utility `caterr` and because we only intend to support English messages, it seemed desirable in the short term to replace the message catalog system with a simple platform independent solution that provided full support for English messages. In the long term we may decide to implement a fuller version.

Our solution was to convert the messages into a static array of message strings, indexed by message number. Aided by a regular expression processor and the fact that the message numbers could be translated directly into array indices, this was a painless process. The new implementation is in

```
Open64/osprey1.0/libcsup/msgnew/mycat.c
Open64/osprey1.0/libcsup/msgnew/mycat.h
```

In order minimize changes to the existing code (allowing for easy uninstallation), we created our own version of the `<nl_types.h>` header which includes the system's version of the header when possible and declares its own symbols when necessary (i.e., on Cygwin). The new header also renames uses of the system catalog calls to our new implementation, ensuring all platforms use the new functions. The new header is

```
Open64/osprey1.0/include/nl_types.h
```

Because this solution hard-codes the message strings in a table, it means that they are compiled into the executable binary and reside in core during runtime, consuming about 170 Kb. Of course, a full replacement of the original catalog system would allow the messages to be stored on disk.

The cvs files involved with this change were tagged with 'tag_gencat_replacment_1' [sic - the typo was not noticed until the command was executed].

3.2 <libgen.h> and basename ()

Cygwin has no version of the system call `basename ()` provided by `<libgen.h>`. It was relatively easy to circumvent this, in one case by using `strstr ()`.

```
Open64/osprey1.0/ir_tools/ir_a2b.cxx
Open64/osprey1.0/ir_tools/ir_size.cxx
```

4 Problems Related to Non-standard C/C++ and Specialized Unix System Calls.

The vast majority of the problems we encountered simply had to do with either non-standard C/C++ code or non-standard Unix calls. The selected list below describes several problems.

4.1 Non-portable Macros

A very common problem accompanying many items below was that although Open64 had been ported to Linux, most conditional compilation macros were written so as to lump every non-Linux system together.

```
#if defined(__linux__)
    /* ... */
#else
    /* ... */
#endif
```

Not only was this sometimes erroneous, it also contributed to some silent errors. For example, in one case `__linux__` was used as a synonym for 'little endian' and when the non-Linux version was used on Cygwin, the big endian code was incorrectly – and silently – used in the compilation. As much as possible, we fixed all such macros to be conditional on the true condition and to, where appropriate, trigger an error on unknown platforms. To illustrate, in situations where the non-Linux code used a SGI-specific system call and the Linux code conformed to standards, we rewrote the macros to be conditional on SGI not on Linux.

4.2 Non-standard C and C++

- Several functions were used in Open64's C++ code that are not in the C++ standard but are in the C99 or Unix standards. (Unix standards are maintained by the OpenGroup.) Examples include `strdup()` which is in the Unix standard but in neither the C99 nor C++ standards and `hypot()` which, though in C99, is not in the C++ standard. In order to conveniently make these standard functions available to C or C++ code we created a standard interface for each function in question. The interfaces are in files whose names are related to where the function's prototype should be found, according to the respective standard. For example, the file `x_string.h` provides the prototype for the `strdup()` replacement. The replacement interface is exactly the same as the respective standard specifies, with the exception of a prefix on the function name that indicates which standard provides it. Continuing the previous example, we defined `ux_strdup()`, indicating that the function is found in the Unix standard. A C function has a different prefix, as in `c_hypot()`. We then changed the Open64 source code to use these standard interfaces.

In most cases, the implementations for these function was trivial. However, if in the future, system and header specific macros might be needed to obtain the appropriate prototypes, this scheme ensures any system specific code is needed only once. For more details see:

```
Open64/osprey1.0/common/util/x_math.[hc]
Open64/osprey1.0/common/util/x_stdio.[hc]
Open64/osprey1.0/common/util/x_stdlib.[hc]
Open64/osprey1.0/common/util/x_string.[hc]
```

- The identifier '`_PTR`' was used as a template parameter – and was being macro replaced by '`void*`' to create a nearly incomprehensible parsing error.

```
template <class ARRAY_Ptr, class T, class _PTR, class REF>
class SEGMENTED_ARRAY_ITERATOR
/* ... */
};
```

The problem with this is that the C++ standard reserves identifiers beginning with '`__`' (double underscore) or with '`_`' (single underscore) and an uppercase for things like macros! We rewrote the template parameters:

```
template <class SA_Ptr, class T, class SA_vt_Ptr, class SA_vt_Ref>
/* ... */
```

- Using `<asm/errno.h>` (unavailable) or `<sys/errno.h>` is non-standard and was replaced by `<errno.h>`.

- The use of `sys_errlist[]` and `sys_nerr` for reporting errors to library calls is non-standard. We eliminated the use of `sys_nerr` and used `strerror()` to obtain error messages.
- We replaced the non-standard use of `fwrite()` with `write()`.
- We removed all uses of the non-standard header `<values.h>`.

4.3 Non-standard Unix Calls

- The use of `sys_siglist[]` for reporting signal error messages is not-standard. Momentum would seem to favor use of `strsignal()`, but this is not yet available on IRIX, MacOS or Tru64. We created `StrSignal()` to standardize the interface.
- The use of `sysinfo()` is not standard. Because the desired information was only the host name, we simply used `gethostname()`.
- The macros `BIG_ENDIAN` and `LITTLE_ENDIAN` are not reliably defined. If not defined, we simply provide our own definition.

`Open64/osprey1.0/common/com/config.h`

- We changed macros so that non-standard `getrlimit64()` would be used only on 64-bit systems. The default is to use the standard `getrlimit()`.
- The macro `BRK_RANGE` is only available on SGI systems. We fixed the macros to only use it on such systems.
- We changed macros to use the SGI-specific `sgidladd()` only on SGI systems; `dlopen()` is now the default.
- We replaced the non-standard and deprecated use of `SIGIOT` with its successor, `SIGABRT`.
- We replaced the non-standard use of the `<wait.h>` header with `<sys/wait.h>`

5 Other Problems

When Open64 was ported to Linux, a set of *system* headers – e.g. `<limits.h>` and `<elf.h>` – was collected and used to help cope with its SGI-specific, non-portable code. They reside in

`Open64/osprey1.0/linux`

Probably, the need for most of these is now much reduced and we would like to eviscerate as many of them as possible. However, because code-cleanup was not the immediate goal, we could only divert some time to this when Open64's duplicate version of `<inttypes.h>` caused problems. Because this header is standard, we gleefully removed it from these Linux-specific headers in favor of the system's version.

Finally, we fixed a non-portable sh script, replacing the non-portable file existence test `test -a file` with either `test -f file` or `test -d dir`.

6 Appendix A

This is a sample GNU Make makefile for the sample code in section 2.3. It assumes the use of GCC's g++ compiler. It can be used on either Linux or Cygwin.

```
#####

INCS = -I.

EXE = test.exe
DSO1 = dsol.dll
DSO2 = dso2.dll

EXE_OBJS = test.o $(DSO1) $(DSO2) test_weak_hack_init.o
DSO1_OBJS = dsol.o
DSO2_OBJS = dso2.o dsol_weak_hack.o

#####

$(EXE) : $(EXE_OBJS)
        $(CXX) -o $@ $^

clean :
        rm -f $(EXE) $(EXE_OBJS) $(DSO1_OBJS) $(DSO2_OBJS)

#####

$(DSO1) : $(DSO1_OBJS)
        $(CXX) -shared -o $@ $^

$(DSO2) : $(DSO2_OBJS)
        $(CXX) -shared -o $@ $^

%.o : %.C
        $(CXX) $(INCS) -c $< -o $@
```