Static Error Checking of C Applications Ported from UNIX to WIN32 Systems Using LCLint

A Thesis
In TCC 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

by

Christopher Barker

March 27, 2001

On my honor as a University student, on this assignment I have neither given nor received unauthorized aid as defined by the Honor Guidelines for Papers in TCC Courses.

Signed_______________________________

Approved_____________________________ Date___________________
Technical Advisor Dave Evans

Approved_____________________________ Date___________________
TCC Advisor W. Bernard Carlson

Abstract

Since personal computer prices have dropped dramatically over the past decade, more companies are feeling a market push to support their software on new platforms such as Microsoft's Windows. Companies have found it to be more cost effective to move older software to the new platforms than to build the software again from scratch. Therefore many companies are attempting to port their old C applications from UNIX to WIN32 operating systems such as Windows NT or 2000, which run on the cheaper PC hardware. When porting C applications from UNIX to WIN32 systems, however, there are a variety of coding issues that can cause the newly ported application not to function correctly. Since it is important for companies to stay competitive in this changing market, these porting issues should be addressed in an efficient manner. By examining documented sources from past software porters and project managers, I compiled a list of coding issues. I used this list in conjunction with LCLint, a popular static checker, to create a set of user-defined annotations in LCLint. These annotations allow LCLint to recognize specific porting bugs and to warn the programmer. This system can be used to reduce costs in porting applications, and give companies increased profits by allowing them to enter a new market more cost effectively. Also, since the system is designed to detect software errors, it is possible that errors could be found that would have lead to catastrophic accidents in mission critical applications or in devices that affect human lives.

Table of Contents

Abstract ii

List of Figures iv

Glossary of Terms v

1. Introduction to the Porting Problem 1
   - 1.1 Why Port Code? 1
   - 1.2 Other Options than Porting 3
   - 1.3 The Perils of Porting 5
   - 1.4 Project Statement 6

2. Debugging Methods 8
   - 2.1 The Limits of Other Debugging Methods 8
   - 2.2 Static Checking 10

3. Finding Porting Issues (Phase 1) 13
   - 3.1 Research Methodology 13
   - 3.2 The Issues 14

4. Designing and Preparation of the System (Phase 2) 17
   - 4.1 Initial Design 17
   - 4.2 LCLint Preparation 19

5. System Implementation (Phase 2 continued) 22
   - 5.1 Warn On Use 22
   - 5.2 Variable States 23
   - 5.3 Global State 26

6. Post Port Testing (Phase 3) 30
   - 6.1 Application Selection 30
   - 6.2 Issues Discovered 31
   - 6.3 Remaining Issues 32

7. Summary and Significance 34
   - 7.1 System Usefulness 34
   - 7.2 System Effects 35
   - 7.3 Recommendations 37

Bibliography 39

Appendix A. Issue List 40

List of Figures

Figure 1. Workstation Market Trend.. 2

Figure 2. Socketstate.xh.. 24

Figure 3. Socketstate.mts. 25

Figure 4. Test1.c.. 27

Figure 5. Sock.c... 28

Glossary of Terms

Annotate - The placing of specific identifiers into source code comments so that a static checker can read and interpret the constraint.

APIs - An abbreviation for application program interfaces. They are a set of software routines that can be used by a program to access system services.

Bug - An error in the source code, or in the design of the source code that results in a piece of software behaving in an undesirable way.

C - A programming language often used to build operating systems. C was used to build UNIX and Microsoft Windows (Wagner 369).

Compile - The process of taking source code and translating it into machine code that a computer can execute.

Execute - The running of code or commands. The computer processor runs the code associated with the file or command that is to be executed.

Operating system - A program that acts as an intermediary between the user of a computer, and the actual machine hardware (Silberschatz 3).

Port - The moving of source code from one machine to another and then compiling it on the new machine.

Priority - A computer can only execute one process's code at a time. A process can be given a certain priority that tells the computer which process should run before another process.

Process - An execution stream of instructions for the computer, in a given state. The state contains all the registers and files that are used by the process.

Socket - A feature in C that allows the programmer to create a connection with another process on the same machine, or a different machine connected through a network, such as the Internet.

Software Life Cycle - The process software undergoes from the original idea for a system to the software's actual use by an end user.

Source code - The programming language version of the computer code before it is actually compiled into machine code.

Static Analysis - The analysis of programs by methodically analyzing the program text. The program is not being executed (Jalote 370).

1. Introduction to the Porting Problem

In the mid 1980's the computer software industry was faced with one of the largest disasters in the computer industry's short history. A machine called the Therac-25 was redesigned from a previous model to run on a new hardware system. After the Therac-25 was released, it caused numerous accidents in the administration of radiation treatments to human patients, including seven deaths. It was later determined that a software bug was the cause of the problem with the Therac-25 (Leveson 18).

The Therac-25 example shows that software bugs can be extremely dangerous. For this reason the utmost care should be taken in the software life cycle to ensure that bugs are prevented or found. This thesis will focus reducing bugs during the porting aspect of the software life cycle.

1.1 Why Port Code?

Over the past decade the computer industry has seen a dramatic change in the number and types of machines sold. In the past, the most commonly used machines were the ones often used by industry or academy for fast and efficient computations. While there were (and still are) large numbers of those computers, most of them used an operating system that was designed to be fair and efficient at allocating resources, but not necessarily easy to use. One of the more common operating systems was UNIX. However as we enter the Twenty-first century this trend is over. Personal workstations, running a Microsoft operating system such as Windows, have become more commonly used. Figure 1 shows that as the computer workstation market expands, Windows

Workstations are experiencing rapid growth, while UNIX systems are relatively constant (Deloitte 4).

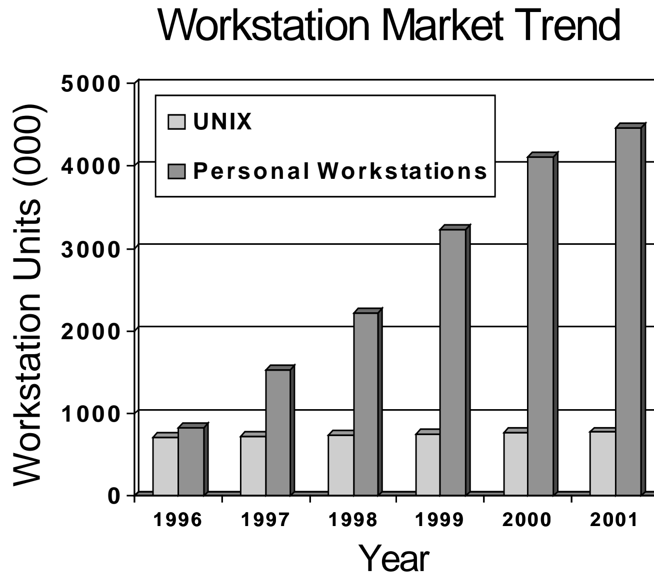

Figure 1. Workstation Market Trend. The change in the number of UNIX and Windows workstations sold. (Deloitte 4).

This changing trend is most commonly attributed to the differences in the two operating systems. UNIX is fast, powerful, allows multiple users at once, can manage multiple tasks simultaneously, and was developed with all the protocols needed for communication through an intranet. UNIX has been around for over 30 years and has established itself as a stable and reliable operating system. In the past, UNIX developers have been mainly concerned with functionality and performance, and spent little time concerned with looks (Wagner 8). While Windows has been around for less then half the time as UNIX, it has appealed to consumers as being more user-friendly than UNIX, and other proprietary operating systems of the past. Windows trades efficiency for fancy little interfaces that make it easy for even small children to figure out. This, coupled with

its increasing stability, functionality, and availability to run on cheaper hardware, has allowed its growth into the market.

Given this market trend it becomes important for companies to offer their software solutions previously written for UNIX environments to the rapidly growing Windows market. These companies could be either software companies that sell there product for revenue, or companies that build there own software for their personal needs. Either way, the most competitive way to get into this new market is to move their already written code over to the new machine. One way to accomplish this task is through porting. Porting is the process of moving the original source code over to a new machine and compiling it. There will usually be some needed changes to the code for it to work on new machine, depending on what the code was doing. However since you are using the same programming language, most of the code will be the same. This allows companies to quickly and easily offer their old products to the new market of consumers.

1.2 Other Options than Porting

Porting code is not the only possible way a company can provide its software product on the Windows architecture. There are two other commonly used methods for providing software to a new architecture, such as Windows. They are starting from scratch and using third party software.

Starting from scratch is, just as the name would suggest, starting over. Basically this process would involve sitting down and rewriting the entire source code on the new platform while trying to incorporate all the new platforms functionality. However, in the software industry you can never really start entirely from scratch. Code reuse is a

common technique used in programming at all stages. Code reuse allows programmers to reuse old and tested code, where applicable, instead of writing new code that hasn't been tested. However even with code reuse, the process of starting over is still lengthy and very time consuming. Imagine rebuilding a large system that may have taken years to build the first time. Once the new product is available to sale, the market may have begun to change again. Even if the code is small enough that starting over will not take years, when completed you will then be faced with a new problem. You would then have two versions of the same software product. You will have the UNIX version and the new Windows version. Then anytime you go to upgrade your product or offer new functionality, you will have two versions to change and then test. This is even more unneeded overhead.

Another method to move software from one platform to another is using third party software. There are several companies that offer software that help software companies in moving their code to a new platform. Nutcracker is a software solution that basically provides a UNIX environment right on Windows platform. Basically you run Nutcracker and it creates a fake UNIX shell where you can run UNIX programs on the Windows machine (Johnston 1). Another popular third party software is Cygwin, which also attempts to give a UNIX environment on a Windows platform. These solutions are relativity new and many people debate over how good they are. There is much discussion currently over the issue of speed and efficiency of these solutions. Time is wasted when code compiled on the "UNIX" platform has to be run through operations that will run on the PC hardware. Also a company must factor in the cost of these third party solutions. The cost of the third party solutions varies from free to rather expensive.

However the free ones are often open source and provide little or no support. Therefore to ensure the system will function correctly with the third party solution, often an expensive product most be purchased. This factor and the possible loss of efficiency make this choice unattractive to many companies.

1.3 The Perils of Porting

Even though porting code manually is usually a better option than starting from scratch or buying a third party solution, it is not easy. There are numerous issues that can arise from a manual port that will leave the new piece of software not functioning like the old one did.

One of the main areas for concern in porting software is using code that is dependant on operating system calls. Most languages, especially C, allow the programmer to make system calls. For example in UNIX, a C programmer may want to get the system time. To do this he would simply call the function G*ettimeofday()* and it would return the time kept by the UNIX operating system. However, if this code were moved to a Windows environment then it would not work correctly. In Windows, a call the function *GetSystemTime()* would need to be made, in order to get the time stored by the Windows operating system. Even then, you are faced with the issue that the UNIX function returns the time in microseconds, and the Windows function only returns the time in milliseconds (Schebert).

An even larger area of concern in porting is when code will compile, but not work like it did on the other machine. In the example above involving system calls to the time functions, the compiler on the Windows machine would probably catch the problem since

it would not recognize the UNIX system call. However, there are situations where the compiler will not catch a potential problem.

In addition to the trouble of system calls, the loss of the original development team is also an issue in porting. With the computer industry and job market being so flexible, when time comes to port an application, many of the original developers may have left the company. Then when a porting issue comes up, the new developers may be faced with a situation where they are not entirely sure what a particular function may be accomplishing. Then in an attempt to make it work on the new system, they change the original functionality, and have actually created a bug by mistake.

1.4 Project Statement

At this point we can see the importance of porting software. It allows companies an affordable and timely means of moving their software from one platform to another, over the other available options. However, we can also see how bugs can be introduced into code after porting as well as how devastating bugs in software could be, as in Therac-25. Therefore this thesis focused on eliminating bugs in C code ported from UNIX to a Microsoft Windows environment.

C source code was chosen since there is an abundance of it developed on UNIX platforms and it is still widely used in software development today. Microsoft Windows, or Windows for short, encompasses several more specific operating systems. This project focused only on ports from UNIX to a WIN32 system. These operating systems, which support 32-bit memory addressing, include Windows 95, 98, NT, and 2000. Most companies that do port applications from UNIX to Windows use Windows NT or

Windows 2000. Both of these operating systems provide more advanced technologies and support more protocols than Windows 95 and 98.

The bug elimination was accomplished in three phases. Phase one will present a list of issues that should be looked for when porting C source code from UNIX to a Microsoft WIN32 environment. Phase two discusses a modification to a static checker, LCLint, to allow it to search for those porting specific bugs found in phase one. Finally phase three will then test the modified static checker on C source code after a port, to see that useful bugs can be found.

2. Debugging Methods

In many mechanical and electrical systems, failures occur over a given time. Aging is the main cause of failure in these other systems. However in software, bugs or failures don't occur due to age. When a failure occurs in software, it means that the bug was in the software since the day it was released, probably earlier. The bug was finally executed under the right conditions at the time the failure occurred (Jalote 5). Because of this difference from other engineering disciplines, the computer science field has had to develop new ways to keep failures from occurring in its software products.

Although my thesis will use static checking as the means for my design to remove ported bugs from software, there are several other commonly used debugging methods in the computer science industry. However, none of these methods were designed to adequately handle bugs in ported software.

2.1 The Limits of Other Debugging Methods

The design of software is broken up into several stages, with all of the stages combined considered the software lifecycle. The main stages of the software lifecycle include the requirements analysis, system design, implementation, testing, and maintenance. Several of these stages have incorporated various techniques to prevent or remove bugs from the end software product.

The requirements analysis and specification phase is the first step in the software lifecycle. Its purpose is to prepare a document that conveys exactly what a client wants a piece of software to do. This document will be used in every stage following its creation.

It will be the basis for the system design and will be used in testing to insure the software is performing like it was originally requested. Because of this, much attention is paid to preventing bugs during this stage. One of the biggest areas of concern is the language of the specification document. Often English is ambiguous or hard to use to specify complex systems correctly. Therefore, languages have been developed just for the development of software requirements. These languages, such as *Z*, allow for formal specifications and for less inconsistency in the document. In theory, this should result in fewer bugs in the end product. However, even if the specification document language helps in the elimination of bugs, this method may not help much with the problem of software porting. Formal specifications remove bugs out of the design, implementation, and testing stages. However porting falls more under the maintenance phase, after the original product has been released. Therefore the activity of porting often involves just the moving of the source code and then a little rework. Rarely will the original specification be considered.

Testing and verification is another area of the software lifecycle that is used to remove bugs. Testing is accomplished by executing a program with a set of test cases as inputs and checking that the outputs are what was expected. While this is successful in removing bugs, it is impossible to remove all bugs with this method alone. Also, the effectiveness of the testing stage relies on the choice of test cases (Jalote 403). The test cases are often selected based on areas of risk determined in the requirements specification. As we have seen, this specification was designed for the original implementation. Therefore after a port, there could be new areas of risk that should be

addressed in the test cases. However if the software still works properly on the old test cases, then the new bugs introduced during porting might be over looked.

In addition to the problem with using old test cases with the ported system, testing has other flaws. Testing itself can be flawed in general, since the test cases are based off the original specification. If this original specification was incomplete or flawed in someway then a test case could be design incorrectly. An error in the specification could result in an output given by the system that is incorrect, even though the specification says it is correct. Therefore, even though an error exists, it wouldn't be found until the end user discovers it, since the software was functioning like the specification specified. If the proper steps were taken in the design of the specification, then it is possible to construct good test cases. However, "devising a set of test cases that will guarantee that all errors will be detected is not feasible" (Jalote 412).

2.2 Static Checking

Static checking is another means of checking programs for bugs. Static checking is usually done with software tools that take source code as input. The static checker then analyzes the code and detects errors or possible errors, and then outputs a report. It is important to note that the source code is not executed during a static check. The static checking tool simply looks at the source code in text format and determines possible issues. Some of the more general things that a static checker will look for include unreachable code, unused variables, and unreferenced labels (Jalote 371).

One of the possible static checkers considered for use in this thesis was Metal. Metal is both a system and a language for static checking. The language provides a

means for a user to tell the system what to look for in the code it is statically checking. Therefore if a user had a set of specific issues he wanted to check for, such as issues when porting C code from UNIX to WIN32 systems, then the user could use the language to "program" Metal to search for these possible bugs (Chou 1).

Metal is under development at Stanford University. The use of Metal for my thesis had several disadvantages. Currently the project is in the development stages and is only used for in-house testing. Therefore it is likely to have little documentation or support, should a problem arise. If Metal was used in my thesis and I ran into either a bug in their code, or a problem in implementation, there was no guarantee that it would be corrected in time for the completion of my work.

Another static checking system is LCLint. LCLint is under development at the University of Virginia. Like Metal, it also provides a system to statically check code, and a means of "programming" it to search for specific issues. LCLint provides user-defined annotations, which is method to allow programmers to define and invent specific annotations, or constraints that should be checked for during the static check (Evans A).

Much like Metal, LCLint is being developed in a university setting. However LCLint is currently used by thousands of people in different settings. LCLint's user-defined annotations are also in development with little documentation, however since they are in development here at the University of Virginia, by my technical advisor, it appeared less risky to rely on LCLint for support than it would have to rely on Metal.

Despite the many different methods of debugging, static checking appeared to be the most suitable method for my thesis. It allows for the discovery of bugs in maintenance phase, and is reliable. Also since one of the goals of this thesis is to save

time and money it is important to mention that static checking is cost effective. "Static analysis is a very cost-effective way of discovering errors" (Jalote 370). It has also been shown that static checking can reduce the time and effort needing during testing.

3. Finding Porting Issues (Phase 1)

In order to modify LCLint so that it can check for bugs in C ports from UNIX to WIN32 systems, the first task was to establish what to have it look for.

3.1 Research Methodology

The first phase of this project began by compiling a list of problems with past ports of C code. This was accomplished by compiling different documented sources of past ports. Computer journals, web sites, and news groups all proved useful in searching for common issues found from past porters. These documented sources were all means for programmers to share their experiences with future programmers. Another source that helped determine possible porting issues was information on the two specific operating systems. By researching the nuances of both UNIX and WIN32 systems, it lead to issues that may arise if the code was ported and was dependant on one of those differences. Personal interviews were another means of gaining knowledge of porting issues with C. I conducted an interview with a student at the University of Virginia that was in the process of moving C source code from UNIX to run on Microsoft's Windows NT. Along with his knowledge I added my own personal knowledge of porting C code. As a summer intern for STR Software in Richmond, Virginia, part of my responsibility was to begin porting their main software product to Windows NT.

Through all these methods of finding possible bugs, a list of issues had to be created. In order for a bug to make my list it had to be either from a respectable source, such as a documented computer journal, or mentioned in several less respectable sources,

such as web sites or news groups, or something I was able to see for myself. The reason for these restrictions is to eliminate false reports and to keep the list of issues as accurate as possible.

3.2 The Issues

The completed list of issues can be broken up into several general categories. These categories include memory management, file input and output, process management, and miscellaneous.

Memory management becomes an issue in porting because of the different ways in which UNIX and WIN32 handle memory. UNIX separates out different types of memory and as a result has separate APIs to handle them. However, this becomes an issue when trying to move over to WIN32, which combines many memory types into a smaller set of APIs (Niezgoda 203). For example the function call in UNIX, shmctl(), performs a shared memory control operations in UNIX, but has no real counterpart on WIN32 systems. Therefore handling of shared memory becomes and important issue when porting.

File input and output is also a concern when moving code from UNIX to WIN32 systems. One of the major concerns in this category is that WIN32 files are handled differently than files are handled on UNIX. Because of these differences, WIN32 systems provide a whole set of special WIN32 APIs as counterparts to the standard C and UNIX file APIs. However WIN32 systems also provide the APIs to most of the standard calls. With WIN32 providing both the standard functions and its own functions, and the fact they cannot be used together, leaves a big issue to be addressed for porters (Digital

9). For example, say that in a standard UNIX C program, a function call is made to *fopen()* to open a file. Later the opened file is passed as a parameter to a function *read()* to read data from that file. Later on this code is ported to a WIN32 system and will run just fine. Then at some point a programmer comes along and adds a WIN32 function called *ReadFile()* and passes it the same opened file. Not only will this compile, it may even work, but Microsoft makes no guarantee that if you pass a file open with a standard API to a WIN32 specific API that it will work. Scenarios like this are common when dealing with files or sockets in ported C applications to WIN32 systems.

In addition to file input and output, process management is another area of concern. One of the most commonly used system calls by UNIX developers is the *fork()* function. It basically creates another identical process. However depending on the next few statements in the program, its counterpart in WIN32 could be completely different. In addition to this issue, there are also other concerns involving process management. For example in UNIX process priority is based on the lowest number being the highest priority. However on WIN32, priority is a set range of numbers, with the highest end being the highest priority. Therefore, it is possible that after a port that the code appear to be working properly, however, the priorities of the processes could be reversed which could be a serious bug, depending on what the process is doing (Digital 9).

The final category that issues were assigned to was miscellaneous. As the name suggests, anything that couldn't fit into a more specific category fell here. One example is the issue of bit size and bit order. On most UNIX systems, the size of an integer declaration in C is 32 bits. Not so standard is the order of these bits. These 32 bits are used to represent a number in binary. The value it represents is based on the most

significant bit, (2 to the 31 power) down to least significant bit (2 to the 0 power). Some UNIX machines are set up so that one end is the most significant bit and other machines are set up so that the other end is the most significant. Then when you get to WIN32 systems they have their own standards about integer size and orientation. Now as long as the code does not do anything that would involve the bit order or size, then there is nothing to worry about after a port. However, if the code is attempting to do logical shifts or forcing values in particular locations on the integer, then this will definitely be an issue during a port to a WIN32 system (Schebert).

4. Designing and Preparation of the System (Phase 2)

The second phase of this project began by selecting the appropriate static checker to use for my design. Once this selection was made, the issues developed in phase one needed to be retooled into a design that could be implemented with the static checker. Also the static checker needed to be setup and prepared for implementation.

4.1 Initial Design

The first design decision to make was which static checker to use in the implementation of my system. There were three possible considerations for the static checker to use in my system, Metal, LCLint, or to build my own from scratch. The third option was quickly removed since it would be very time consuming and was well out of the scope of this project. Metal and LCLint both have the capabilities necessary for the completion of this project. Therefore it made no sense to reinvent the wheel when two reasonable alternatives were available.

As discussed in chapter 2, both Metal and LCLint have the drawbacks of having little documentation. However since LCLint was in development here at the University of Virginia, by my technical advisor, and Metal was under development at Stanford University, LCLint made the most reasonable choice. Using LCLint as my static checker would give me direct contact to one of the developers of the user-defined annotations, which would be used heavily during implementation. Thus reducing the risk of running into a bug in the static checker and not being able to continue. It also gave me some freedom to request certain enhancements to the static checker for my own use, which I would not have been able to do if I used Metal. Another aspect in favor of LCLint was

that it is used by thousands of users while only a few people have used Metal. Using LCLint would allow for my research and design to gain the most exposure. Therefore due to LCLint's capabilities, its available support, and its wide use, it was chosen as the static checker for use in my system.

Once LCLint was selected, the next step was retooling the issues found in phase one in a way that they could be programmed into LCLint's user-defined annotations. One way to use these annotations is to assign a state to a variable when it is created. Then annotate how that variable's state changes when passed into different functions or after different functions are called. Then if the programmer attempts to pass a variable to a function, and that variable is in the wrong state, then LCLint will catch it and warn accordingly. For example, one of the issues discovered in phase one was that in UNIX, socket connection are closed by a call to *close(int)*. However, in WIN32 systems, the *close(int)* only closes files, while *closesocket(int)* has to be called to close sockets. Therefore in order for LCLint to catch the mistake of a programmer attempting to close a socket using the file close function call, I would assign a state to any variable that was a socket. Then if that state was passed into the *close()* function, it would be an illegal state and LCLint would inform the programmer accordingly.

While some of the issues from phase one were easy to incorporate into LCLint, others were more difficult. Some of the issues were impossible to define for LCLint and therefore had to be left out of the scope of the project. Several of these issues involved the passing of buffers to specific functions and checking the size of those buffers. For example, one of the issues was that the *CreateProcess()* function on WIN32 systems would only take up to 1024 characters being passed into it. However it's counterpart on

UNIX systems had no such limit. So a common porting issue would be the truncation of a long string being passed into *CreateProcess()*. However at the time of this thesis, LCLint was not currently capable of checking for buffer length or buffer overflow. Therefore even though this issue is important to consider in the porting of C code from UNIX to WIN32 systems, I was unable to incorporate it into the system. Buffer checking is currently being developed into LCLint's capabilities, therefore this issue and a few others that were left outside the scope of this project for similar reasons, could be added at a later date.

4.2 LCLint Preparation

Once LCLint was chosen as the static checker for this project, and the issues from phase one where retooled into a way that they could be implemented into LCLint, the next step was preparing LCLint for implementation.

In order for LCLint to work effectively it needs to have a library of function headers for all the system provided functions. By system provided functions I mean all the standard C functions (i.e. printf(), read(), malloc()) and all the WIN32 system calls (i.e. CreateProcess(), GetSystemTime(), CreateThread()). LCLint comes standard with the UNIX, ANSI, and POSIX libraries but currently doesn't have aWIN32 library. Therefore the first step to preparing LCLint for my implementation was to create a WIN32 library with the necessary WIN32 system calls and standard WIN32 C function calls.

Based on the modified issues, I began by searching the windows include files that come with the distribution of Microsoft's Visual C/C++. These include files would

contain all the needed function declarations for all the WIN32 system calls as well as the standard C functions. The search consisted of all functions that would be used or affected by the issues retooled earlier in phase two. Any file that contained a function declaration that was needed was incorporated into the WIN32 library. For example, the function *closesocket(int)* which was discussed earlier was needed for an issue. That function declaration was located in the file, winsock2.h. Therefore when compiling all the files needed for the WIN32 library, winsock2.h was included. While a complete WIN32 library would be useful and will be the overall goal of LCLint developers, its creation was out of the scope of this thesis.

There were several reasons why only the needed files were included in the creation of the WIN32 library rather than creating an entire library, with every WIN32 system call. One of the issues was speed. Every time LCLint is executed, the WIN32 library would need to be read. I determined it would be a waste of time to read in the entire library when only a handful of the actual files would be needed for my system. Another area of concern was problems with LCLint compiling the WIN32 code. There were several instances throughout the creation of the library where LCLint was unable to parse the WIN32 code. Some of the WIN32 code had to be modified, renamed, or even commented out after the problem was determined, in order for LCLint to function correctly. Therefore the inclusion of unneeded files would only result in more time and work that would be unneeded to complete this thesis, and therefore out of the scope of the thesis.

Some other modifications were made to the WIN32 header files to make them more useful for LCLint. Many of the defined types were modified to include annotations

to allow LCLint to check for incorrect usage of user-defined types. Also modifications were made to all the function headers. Microsoft had named all the variables passed into functions in the headers, which is a bad practice and doesn't allow for macros to be named the same. Therefore all variables passed into functions were tagged with a *p_* in front to avoid this issue.

Once LCLint had a working WIN32 library, which included all the needed files and function declarations involved in my thesis, LCLint was ready for the implementation of my design. Chapter 5 will continue with phase two, with more detail into the implementation of the system.

5. System Implementation (Phase 2 continued)

The implementation of the system was a big task. In order to simplify this task the implementation was broken down into several smaller stages. These stages involved programming in different issues based on how they would have to be implemented into LCLint.

5.1 Warn On Use

The first category of implementation was warn on use. This involved all the issues that were of serious importance to porters, but that a static checker could not determine if an actual error had occurred. The implementation was done by adding annotations to the WIN32 library to necessary functions. An annotation to the library would look like the following /*@warn unixtowin "message"@*/. The surrounding slash and star are comments in C and C++. This way when a compiler reads the code, this line would not interfere with the actual code. The "at" symbol surrounds the statement telling LCLint that this is an annotation. Once LCLint sees this, then it will read in the warn command, a flag (unixtowin), and finally the message in quotes. Therefore, after LCLint reads in this library, it will then warn the programmer with the provided message anytime the particular function, with the annotation, is used. The programmer can suppress this warning after the issue is addressed by providing LCLint with the flag value at the command line.

For example, one of the implemented warn on use issues was involving the function *SetProcessPriority().* The issue here is that on WIN32 systems the value passed to this function can be a set range of values, with the highest number making the process

the highest priority (Digital 9). However, in UNIX process priority is set with the lowest number being the highest priority. This issue was addressed by adding a warn on use annotation to the WIN32 library in the declaration of the function *SetProcessPriority()*. The reason a warn on use annotation makes sense here is that we cannot tell if the programmer really wants this process to be a high or low priority. All we know is that this is a common porting issue, so we can warn the programmer of the problem, he can investigate it, and then suppress the message after he addresses the issue.

5.2 Variable States

Once all the warn on use constraints were added, the next step was the implementation of the variable state constraints. These annotations where briefly discussed in chapter 4. By creating *.xh* files, state values are assigned to variables when they are passed to functions or after functions are called. Then *.mts* files define a set of rules, allowing states to transition to other states, but then raising errors if the wrong state is passed to a function. Then when LCLint reads in the *.xh* and *.mts* files it will know how to handle the variable states.

Lets reconsider the issue discussed in chapter 4 involving the closing of files and sockets. In UNIX the function *close(int)* will close both open files and open sockets. However, on WIN32 systems the *close(int)* only closes files, and the *closesocket(int)* only closes sockets. A likely porting issue is trying to use the *close(int)* function on WIN32 systems to close sockets. Using a variable state checks for this issue. Figure 2 shows an excerpt from socketstate.xh, which is the file that defines the annotations made to the WIN32 library to handle this issue.

```
#include "winsock2.h"
WINSOCK_API_LINKAGE
/*@socketopen@*/
SOCKET
WSAAPI
socket(
int p_af,
int p_type,
int p_protocol
);

WINSOCK_API_LINKAGE
/*@socketopen@*/
SOCKET
WSAAPI
accept(
/*@socketopen@*/
SOCKET p_s,
struct sockaddr FAR * p_addr,
int FAR * p_addrlen
);

WINSOCK_API_LINKAGE
int
WSAAPI
closesocket(
/*@socketopen@*/
SOCKET p_s
)
/*@ensures socketclosed p_s@*/;

_CRTIMP int __cdecl _close(/*@socketna@*/ int);

_CRTIMP int __cdecl close(/*@socketna@*/ int);
```

Figure 2. Socketstate.xh. The LCLint file that defines the state of variables in WIN32 functions. Annotations inside of functions define the state that the variable should be when the function is called. Annotations outside of functions define the state of the return variable.

Anytime a socket connection is opened (this can be accomplished by a call to several different functions) the variable assigned the socket value is given a state "socketopen". The function *closesocket(int)* was annotated so that the state "socketopen" should enter into it, and the state of the variable after the function call is "socketclosed".

The function *close(int)* was annotated to allow only variables of the state "socketna" to pass through.

```
state socketstate
   context reference /*type SOCKET and int */
   oneof closed, open, na, dc
   annotations
     socketopen reference ==> open
     socketclosed reference ==>closed
     socketna reference ==> na
     socketdc reference ==> dc

   transfers
     open as closed ==> error "Open socket passed as
     closed"
     closed as open ==> error "Closed socket passed as
     open"
     open as na ==> error "Sockets not allowed to be
     passed to this func"
     closed as na ==> error "Sockets not allowed to be
     passed to the func"
     dc as open ==> error "Possible illegal passing of
     a socket"
     dc as closed ==> closed

   losereference
     open ==> error "Lost reference to open socket"

   defaults
     reference dc
end
```

Figure 3. Socketstate.mts. The LCLint file that defines the rules of state transitions for the "socketstate" variable state definition.

Figure 3 shows the socketstate.mts file, which defines the rules on the possible transitions through states. LCLint, after reading in these two files, can now check to make sure that sockets are closed properly. If a socket is opened it will now be assigned a state of "socketopen". If the function *close(int)* is called with a "socketopen" state passed to it, the rules say to raise an error with the message "Sockets not allowed to be passed to this func". If the programmer forgets to close the socket, the lost reference rule

in the socketstate.mts file will raise an error, which prevents the program from exiting with a socket in the open state.

There are two important things to note about this implementation. First, it will signal the programmer should he accidentally try and close a socket incorrectly. The second thing to note is that this method of implementation has some good side effects. It also will signal the programmer if he tries to send a closed socket to the *closesocket(int)* function. Also, it only allows open sockets to be passed to the all read, write, and other functions allowed to perform operations on sockets. So not only does this implementation solve the porting problem, it also offers numerous other checks for free.

5.3 Global State

The final method of implementation used was the assigning of global state. This works similar to variable state but instead of assigning a state to a local variable, a state is assign globally to the whole system. The purpose of this is to ensure that proper initialization procedures are called before particular functions or operations can be called.

On UNIX systems, C programs can open sockets by a simple call to the function *socket()*. On WIN32 systems, sockets are also open using the same call to the function *socket()*. However, WIN32 systems first require the function *WSAStartup()* be called to initialize the sockets, before any calls to socket functions. If this function is not called, the compiler will not catch it and Microsoft makes no guarantee as to how the sockets will function. Therefore when porting from UNIX to WIN32, it is important that this issue be addressed or a bug may be introduced into the program.

By defining a global state, we can prevent the calling of socket functions before a call to the initialization function. Until the initialization function is called the global state will remain "sockets_uninitialized". If a socket function is called while in this state an error will be raised informing the programmer that he is not in the correct state to call socket functions. Once the initiation function is called, socket functions will operate as they normally would.

5.4 Self-tests and System Limitations

During the various implementation stages each design was constantly being tested and redesigned where needed to insure correct functionality. One of the first tests run was to see how useful the warn on use checks were. LCLint was tested on a small program, excerpt shown in Figure 4, which contained four different references to the function call *select()*, which is a warn on use issue.

```
fd_set test_select() {
     //calls select() function to test for warn on use
     //issues.
     return select(read_fd, write_fd, NULL, NULL);
     }
```

Figure 4. Test1.c. A function from a test program written to test the warn on use implementation of the function select.

Only one of these references was actually a function call to *select(),* the rest just being comments or something else that should not have been noticed by LCLint. Even though *grep*, which is a popular text searching tool for programmers, found all four instances of *select()* in the program, LCLint only warned on the one actual function call

to *select(),* which was embedded in a return statement. Thus showing the usefulness of LCLint and the warn on use checks.

Each of the variable state implementations where tested individually and as a whole to see that they caught the specific errors they were designed to catch. Figure 5 below shows a test program used with the "socketstate" implementation shown in Figures 2 and 3. The comments show when errors should be raised. These self-tests showed two important areas of concern with the implementation.

```
#include <winsock2.h>

int main() {
        int a,b,c,d,e,f,g;

        closesocket(a);         //error dc as open
        b = accept(a, NULL,NULL); //error closed as
        open
        select(d,NULL,NULL,NULL,0); //okay, but
        warnuse
        send(b,NULL,NULL);      //okay
        close(b);       //error open as na
        close(d);       //error dc as na
        c = socket(NULL,NULL,NULL);  //okay
        closesocket(c);         //okay
        return 0;       //error lost ref to open b
        }
```

Figure 5. Sock.c. The C test file used to test the "socketstate" implementation discussed earlier. Notice the comments report when errors should occur.

The first concern was parameter passing. Since LCLint performs a static check, it cannot trace to actual flow of the program. This means LCLint examines each function and procedure of the program individually, giving no attention to the order functions are called. This raises an issue when considering the implementation of variable states. If a variable is assigned a value in one procedure, then that variable is passed to another function the value of that variable will be lost when LCLint checks that function.

Because of this, LCLint may raise unnecessary errors or even miss errors if this issue is not addressed. Fortunately this problem can be easily solved. Adding annotations to the functions in the program can ensure that each variable keeps the correct state when it is passed from function to function, just like the way each function is annotated in the actual WIN32 library.

6. Post Port Testing (Phase 3)

Once LCLint was completely implemented and had been initially tested to see that it worked as designed, phase 3 began. Phase 3 was the process of using LCLint on a real application and interpreting the results. While the main goal of my thesis was to design and build the system, testing to see that it works is also important. Extensive testing would be needed in order to determine the exact usefulness and effectiveness of the system, however that is beyond the scope of this thesis.

LCLint with the new annotations was tested on an application that had already been ported from UNIX to WIN32. It was believed this approach should yield more interesting results about the system than a test on an application that had not been ported yet. Although, this system could be used on applications prior to porting and those results could be used in the porting decision.

6.1 Application Selection

The application selected for the test was a ping program that was written in C and ported to Windows NT. This application was selected for several reasons. First, it was a C program that had been ported and the source code was available. Second, the program was a relatively simple program, with only several hundred lines of code in two files. It was important that the code not be too large otherwise I may not be able to determine what was going on if an error did arise. Another factor that contributed to the ping application being selected was that it used sockets quite heavily. Since many of the issues were socket related, it made a good choice since it had the better chance of triggering an issue, than a program that did not use sockets.

The testing of the ping application resulted in just less than fifty reported errors from LCLint. Of these errors, only seven were related to porting. Many of the remaining errors were related to issues that LCLint commonly checks for including, loss of return values, incorrect assignments, and incorrect passing of NULLs to functions. Of the seven reported porting errors, two of those were deemed useful with respect to the ping application. The remaining five errors pointed out some issues with the checking system. While five unwanted errors may seem like a lot, by examining these errors we will see that they were not unwanted errors, but the result of our approach to solve the problem.

6.2 Issues Discovered

The first issue that was found in the ping application was the incorrect use of an abstract type *SOCKET*. During the preparation of the WIN32 library for LCLint, several of the Windows types were annotated to prevent them from being used as their base type. *SOCKET* types are basically integer types, however they are given the name *SOCKET* to represent integers that are socket descriptors. This way, even though a *SOCKET* type could be used as an integer, as in arithmetic operations, it should only be used in socket operations. By annotating this in the WIN32 library, LCLint checks to make sure that *SOCKET* is being used correctly. However in the ping application, LCLint found an instance were a *SOCKET* type was being used as an integer, because an integer value was being arbitrarily assigned to it. This should not be allowed since socket functions will only work on assigned socket descriptors, not just any arbitrary integer.

The second issue that LCLint raised in the checking of the application was the lost reference to an open socket. This was found because at one point a socket had been

opened and the LCLint found a way to exit the program without closing that socket. At first inspection of the ping application, a call to the function *closesocket()* was found. However a more detailed search finally lead to the path LCLint had discovered. It was possible to open a socket then later hit an error and exit, without first closing the open socket.

6.3 Remaining Issues

The other issues found by LCLint turned out to be important, not because they found bugs in the ping application, but because they pointed out an error in the testing approach of this system. LCLint reported two more instances that the program exited without closing an open socket. In addition LCLint also reported two instances of illegal transfers from a "dcstate" to an "openstate". This meant that the ping application was trying to pass a variable as an open socket, when it had not been opened. After inspecting the ping code further, it was determined that these reports were incorrect and that the problem was with parameter passing from one function in the ping application to another. Since LCLint was checking the program statically, it has no way of knowing when it gets to one function, if the variables passed into it are already in a particular state. Therefore it treats them as a new declaration and they get assigned the default state. This is why the two errors about the illegal transition came up. Early in the ping application a socket was opened and then a function was called to send some data. That function was passed the open socket and used the send function to send out data. The send function has been annotated to only use open sockets, however when LCLint inspected that function, it assigned the default state to the variable. Therefore the result led to the

errors. Then when it exited that function, it did not close the open socket because it is closed else where in the program therefore that led to two more reported errors.

While this issue with parameter passing resulted in four unwanted errors it was also easily solved. By adding annotations to the function declarations themselves, then the correct functionality was achieved. For example, by going back to the send function in the ping application, an annotation was added to the function saying that only open state sockets should be passed into the function. Then when LCLint examines the function it will know that the socket in that function is open and will not raise the error about the illegal transfer. Also since that function now only takes open sockets, if that function is called anywhere in the program and is not passed an open socket, then an error will be raised. Therefore even though the ping application raised those four unwanted errors, two them were because the testing of the ping application was approached incorrectly. Once the annotations were added to the ping application, just like in they were added to the WIN32 library, these errors were no longer raised.

The one remaining issue that LCLint raised was a warn on use for the select function. The select function was used by the ping application and therefore LCLint warned me about the timing issues and the socket error issue. However, neither of these concerns were relevant in this application. Therefore I used the flag and suppressed the warning and continued checking.

Overall the ping application was very useful in the checking of LCLint with porting annotations. Not only did it show that LCLint could find issues in a real application, but it also showed the importance of annotating the source code so that LCLint can correctly determine what the programmer is trying to accomplish.

7. Summary and Significance

Now that we have seen how this debugging system using LCLint was built, and how it works, it is also important to show how useful the system will be, and how its use can affect the world around us. Then finally, I will make recommendations for future research and testing in this area.

7.1 System Usefulness

Through the three phases discussed in the previous chapters, it can be shown how LCLint, with the new modifications, can be a useful tool in handling porting issues. Phase one showed that there was numerous issues that arise when porting C code from a UNIX system to a WIN32 system. Phase two then showed that a system could be designed and implemented to check for the bugs in an efficient manner. Finally in phase three we saw that the modified LCLint was capable of detecting bugs in software.

Therefore this system can be viewed as useful in two distinct ways. Before a port, LCLint could be ran on the code that will need to be ported. Based on the issues that are raised, the programmers could come up with the best plan to address these issues. Also if the cost of third party software is less than the estimated time of redesigning the code due to the raised issues, LCLint may also help in the decision of buying a third party solution rather than attempting the port at all. If the code has already been ported, LCLint could help in a different way. After a port, when getting ready for system testing, LCLint could be useful in helping to eliminate bugs that could be missed in the testing phase. Not only has the software developing community stated that static checking is a cost-effective way of discovering errors, they also claim that static analysis can also spot the actual error

itself, unlike testing which usually spots the presences of an error (Jalote 370). So since system testing is large part of the software development process, any method to help reduce this time would obviously reduce costs. Thus allowing companies to move into new markets faster with more reliable products.

7.2 System Effects

Not only can LCLint with the modifications check for porting issues, but it can also affect various aspects of our society. This system can have effects at the local level here at the University of Virginia, at the business level, and to society as a whole.

This system using LCLint is the first use of Professor Evans's LCLint, with the newly developed user-defined annotations. During the course of the project, as well as by looking at the results, Professor Evans will be able to use this project as a way of weighing the success of this new feature. I found it relatively easy to add my own annotations to LCLint, to give LCLint the functionality to check for specific issues in ported C code, and this was with little or no documentation. Therefore, Professor Evans may determine that this new feature is useful and that his overall product has been greatly enhanced by this new feature. He may decide to add documentation and release this latest feature and announce the new release accordingly. Since LCLint is relatively widely used, my results could then be distributed to thousands of LCLint users.

Another way my thesis could affect society is through software companies. Currently after a company decides to port its product from a UNIX environment to a WIN32 environment, the most common way to guarantee it is working correctly is by testing. So after a possibly tedious port, more time would further need to be spent testing

the software, as though it had just been created. With the availability of this static checker that can check for specific porting issues, this time spent testing can be decreased. This decreased time usually means less labor and the quicker turn around for products, thus increasing revenue.

Another way this project could impact the corporate world is by its affect on third party providers. If a company does decide to use LCLint with the annotations for porting, the company may be able to determine if the cost of the third party solution out weighs the cost of a manual port. If more companies find that third party solutions are the better choice, than this system could lead to the increase of revenue for third party solutions. If this system instead can provide more confidence in the functionality of the system and reduce the overall cost of a port, then less companies may select to use a third party vendor. This result could lead to lost revenue for third party solutions, and ultimately to layoffs or restructuring of their products. The actually impact is hard to say because the results of this system will vary depending on the companies source code.

The biggest area of impact this project can affect is human life. While this project focused on a specific aspect of static checking and porting, it ultimately related to producing fault tolerant software. Static checking is just another means programmers use to verify and test software. So since this project focused on developing a system that checks the functionality of software, it can be shown that this system can lead to saving human lives.

There is no disputing that computers are becoming increasing more important in our daily lives. Software systems control the planes we fly in, weapons of mass destruction, and even things such as heart pace makers. The more we rely on these

computers in our daily lives, the more likely a software bug or failure could result in human injury or death. A popular example of this can be seen by the Therac-25, as discussed at the beginning. This device was designed to give radiation doses to cancer patients, however, due to a software bug it killed 7 people and injured more (Leveson 18).

Phase one of this project showed that bugs are introduced during porting. Phase two and three showed how LCLint with annotations for porting could find these bugs. Therefore if this system is used on software that effects human lives, it could be possible for this system to find a bug that could have lead to a possible human casualty. Therefore this system could ultimately help increase software reliability in software running on devices that effect human lives, so that disasters like Therac-25 don't happen again.

7.3 Recommendations

Given the possible significance and usefulness of this project, there may be researchers that may want to continue with my research. There are two main areas that further study could be warranted.

The biggest area of future research would be the including of additional constraints. If more porting bugs are found, or if some more of the ones found in phase one of this project can be added to LCLint's capabilities, then this would only improve LCLint's effectiveness. Even though the system should be measured by the quality not the quantity of the bugs found, the more issues that LCLint is capable of finding, the more useful the system may be to companies and society as a whole.

Another area of future research would be incorporating different languages. C was chosen because there was an abundance of it written on UNIX machines, which was still useful. No doubt there are other languages that a static checker capable of checking porting issues would benefit. If this task was undertaken, not only would new constraints need to be found for that particular language, a different static checker would need to be used since LCLint currently only works with C code.

Bibliography

Chou, Andy, and Dawson Engler. *Metal: a language and system for building lightweight, system-specfic software checkers, analyzers, and optimizers.* Available upon request: acc@cs.stanford.edu, 2000.

Deloitte & Touche Consultant Group. *Deploying Windows NT in Technical Workstation Environments.* Online. 1997. Available: http://www.microsoft.com/ntworkstation/technical/WhitePapers/MigrateUnix.asp.

Digital Equipment Corporation, *Digital UNIX and Windows NT Interoperability Guide*, USA, 1996. Digital Equipment Corporation. Available: http://wint.decsy.ru/du/Digital/Unixnt/index.htm

Evans, David. "Annotation-Assisted Lightweight Static Checking." Position Paper for *The First International Workshop on Automated Program Anaylysis, Testing, and Verification*. Available: http://lclint.cs.virginia.edu/icse-position.html

Evans, David. *LCLint User's Guide*. Online. May 2000. Available: http://lclint.cs.virginia.edu/guide/.

Giguere, Eric. "Porting C Programs." *Computer Language* February 1988: 75 – 78.

Glass, David. "Porting UNIX Applications to DOS." *Dr. Dobb's* November 1991: 68-76.

Jalote, Pankaj. *An Integrated Approach to Software* Engineering. New York: Springer-Verlag New York, Inc., 1997.

Johnston, Stuart. "UNIX to NT Hassle-free." *Information Week Online* February 22, 1999. Available: http://www.informationweek.com/722/unixnt.htm

Leveson, Nancy, and Clark Turner. "An Investigation of the Therac-25 Accidents." *IEEE Computer* July 1993: 18-41.

Niezgoda, Steve. "Charting the Uncharted." *Byte* October 1994: 203- 204.

Schubert, Brenden, Third Year attending University of Virginia, Charlottesville, VA. Personal Interview. 04 October 2000.

Silberschatz, Avi, and Peter Galvin. *Operating System Concepts.* New York: John Wiley and Sons Inc, 1999.

Wagner, Bill. *The Complete Idiot's Guide to UNIX.* Indianapolis: Que Corporation, 1998.

Appendix A. Issue List

The following is the list of issues discovered during phase 1 of this thesis. Each issue includes a description of the problem, files involved, and the method of solving the problem, if applicable.

Issue 1: *closesocket() vs close()*

Description: In UNIX, a socket is closed by a call to the function *close()*. In WIN32 systems, the function *closesocket()* must be called, however, the function *close()* does exist with different functionality.

Files Involved: winsock2.h, io.h

Solution: Solved by creating a variable state called socketstate. Anytime an integer or socket type was declared it was given a state of *dcstate*. If the variable was assigned a value from a function that would result in an open socket (*socket(), accept(), WSASocket())* then the state was changed to *socketopen*. If a variable with the state of *socketopen* was passed to the function *close()* then an error would be raised. If the program could exit without a call to *closesocket()* and a variable was in the state *socketopen* then an error was raised.

Issue 2: Standard C APIs vs. WIN32 APIs for Sockets

Description: WIN32 systems provide both their own functions to create and use sockets, as well as the standard methods available in C on UNIX systems. It is possible to create sockets using one method and then use the socket with functions from the other method. However, Microsoft does not guarantee correct functionality if this is done, so it is important to be consistent.

Files Involved: winsock2.h, io.h

Solution: Solved by creating a variable state called apistate. Anytime an integer, handle, or socket type was declared it was given a state of *dcstate*. If one of the numerous socket functions where called, the state was changed to *unixapi* or *winapi* depending on which was used. Then if the variable was passed to a function of the opposite type, an error would be raised.

Issue 3: Standard C APIs vs. WIN32 APIs for Files

Description: WIN32 systems provide both their own functions to create and use files, as well as the standard methods available in C on UNIX systems. It is possible to create or open files using one method and then read or write to the file using functions from the other method. However, Microsoft does not guarantee correct functionality if this is done, so it is important to be consistent.

Files Involved: winsock2.h, io.h, stdio.h, winbase.h

Solution: Solved by creating a variable state called apistate, in conjunction with issue 2. Anytime an integer, handle, or socket type was declared it was given a state of *dcstate*. If one of the numerous file functions where called, the state was changed to *unixapi* or *winapi* depending on which was used. Then if the variable was passed to a function of the opposite type, an error would be raised.

Issue 4: Polling files

Description: In UNIX, files as well as sockets can be polled to see if data is ready to be read or written. On WIN32 systems, you cannot poll files, and if done the results are arbitrary.

Files Involved: winsock2.h, io.h stdio.h, stat.h, types.h

Solution: Solved by creating a variable state called

Issue 5: Error Checking of Sockets

Description: In UNIX, errors or hang-ups on sockets are detected and reported by placing the socket file descriptor in the error set after a *poll()* or *select()* function is called. On WIN32 systems, errors are also reported by signaling a that a socket is ready to read, and then reading zero bytes. Therefore relying on select to detect hung-up sockets is not a good policy on WIN32 systems.

Files Involved: winsock2.h

Solution: A warn on use annotation was added to the select function. Whenever the select function is used, it warns the programmer of the difference and he can make any needed changes and then suppress the message.

Issue 6: Initiation of Sockets

Description: In UNIX, sockets can be created with no initializing, but on WIN32 systems, sockets must first be initiated by a call to the function *WSAStartup()*.

Files Involved: winsock2.h

Solution:

Issue 7: Timing of Timeouts

Description: In UNIX, the poll function timeouts after the supplied time in milliseconds. On WIN32 systems the select function timeouts after the supplied time in seconds and milliseconds. Easy to get confused and have timeout to long.

File Involved: Winsock2.h

Solution: A warn on use annotation was added to the select function. Whenever the select function is used, it warns the programmer of the difference and he can make any needed changes and then suppress the message.

Issue 8: *CreateProcess()* Parameter Passing

Description: Only 1024 characters are allowed to be passed to the function *CreateProcess()* on WIN32 systems. It's counterpart on UNIX, *fork(),* has no restriction.

Files Involved: winbase.h

Solution: Warn on use annotation added to the function. Whenever the function is used, it warns the programmer of the difference and he can make any needed changes and then suppress the message. A better solution can be incorporated into LCLint once overflow buffer checking is implemented.

Issue 9: *CreateProcess()* with Environment Variables

Description: Environment variables with the "$" passed to the function *CreateProcess()* on WIN32 systems will not be expanded like it will with its counterpart on UNIX.

Files Involved: winbase.h

Solution: Warn on use annotation added to the function. Whenever the function is used, it warns the programmer of the difference and he can make any needed changes and then suppress the message.

Issue 10: Bit Size and Bit Order

Description: The sizes and ordering of bytes varies from UNIX machines to WIN32 machines. Any code that depends on bit size or order could result in a bug after a port.

Files Involved: None

Solution: No solution implemented.

Issue 11: Priority

Description: On UNIX systems, process priority is set with the lowest number being the highest priority. On WIN32 systems, the highest priority is the highest number in a set of values. Therefore, after a port is likely the programmer may inadvertently switch the priority level.

Files Involved: **winbase.h**

Solution: Warn on use annotation added to the functions *SetProcessPriority()* and *SetThreadPriority()*. Whenever the function is used, it warns the programmer of the difference and he can make any needed changes and then suppress the message.

Issue 12: File naming issues

Description: In UNIX filenames and paths can have some of the various symbols in them that are not allowed on WIN32 systems: (<,>,|,/,:,\).

Files Involved: None

Solution: No solution implemented.

Issues 13: Inodes

Description: WIN32 systems does not support inodes although it has all the necessary functions and data types defined, like UNIX.

Files Involved: types.h, stat.h

Solution: No solution implemented.

Issue 14: Text vs. Binary Files

Description: WIN32 systems distinguish between text and binary files, however UNIX systems do not. Therefore file open commands on UNIX machines only need "r" passed to it to open either binary or text files. However, on WIN32 machines if only the "r" is passed to a file open command, only text files will be read.

Files Involved: stdio.h, io.h

Solution: No solution implemented.