Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Native POSIX Threads Library (NPTL) Support for uClibc.

Steven J. Hill
Reality Diluted, Inc.
sjhill@realitydiluted.com

Abstract

Linux continues to gain market share in embedded systems. As embedded processing power increases and more demanding applications in need of multi-threading capabilities are developed, Native POSIX Threads Library (NPTL) support becomes crucial. The GNU C library [1] has had NPTL support for a number of years on multiple processor architectures. However, the GNU C library is more suited for workstation and server platforms and not embedded systems due to its size. uClibc [2] is a POSIX-compliant C library designed for size and speed, but currently lacking NPTL support. This paper will present the design and implementation of NPTL support in uClibc. In addition to the design overview, benchmarks, limitations and comparisons between glibc and uClibc will be discussed. NPTL for uClibc is currently only supported for the MIPS processor architecture.

1 The Contenders

Every usable Linux system has applications built atop a C library run-time environment. The C library is at the core of user space and provides all the necessary functions and system calls for applications to execute. Linux is fortunate in that there are a number of C libraries available for varying platforms and environments. Whether an embedded system, high-performance computing, or a home PC, there is a C library to fit each need.

The GNU C library, known also as glibc [1], and uClibc [2] are the most common Linux C libraries in use today. There are other C libraries like Newlib [3], diet libc [4], and klibc [5] used in embedded systems and small root file systems. We list them only for completeness, yet they are not considered in this paper. Our focus will be solely on uClibc and glibc.

2 Comparing C Libraries

To understand the need for NPTL in uClibc, we first examine the goals of both the uClibc and glibc projects. We will quickly examine the strengths and weaknesses of both C implementations. It will then become evident why NPTL is needed in uClibc.

2.1 GNU C Library Project Goals

To quote from the main GNU C Library web page [1], “The GNU C library is primarily designed to be a portable and high performance C

library. It follows all relevant standards (ISO C 99, POSIX.1c, POSIX.1j, POSIX.1d, Unix98, Single Unix Specification). It is also internationalized and has one of the most complete internationalization interfaces known." In short, glibc, aims to be the most complete C library implementation available. It succeeds, but at the cost of size and complexity.

2.2 uClibc Project Goals

Let us see what uClibc has to offer. Again, quoting from the main page for uClibc [2], "uClibc (a.k.a. μClibc, pronounced *yew-see-lib-see*) is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc. Porting applications from glibc to uClibc typically involves just recompiling the source code. uClibc even supports shared libraries and threading. It currently runs on standard Linux and MMU-less (also known as μClinux) systems..." Sounds great for embedded systems development. Obviously, uClibc is going to be missing some features since its goal is to be small in size. However, uClibc has its own strengths as well.

2.3 Comparing Features

Table 1 shows the important differentiating features between glibc and uClibc.

It should be obvious from the table above that glibc certainly has better POSIX compliance, backwards binary compatibility, and networking services support. uClibc shines in that it is much smaller (how much smaller will be covered later), more configurable, supports more processor architectures and is easier to build and maintain.

The last two features in the table warrant additional explanation. glibc recently removed `linuxthreads` support from its main development tree. It was moved into a separate ports tree. It is maintained on a volunteer basis only. uClibc will maintain both the `linuxthreads` and `nptl` thread models actively. Secondly, glibc only supports a couple of primary processor architectures. The rest of the architectures were also recently moved into the ports tree. uClibc continues to actively support many more architectures by default. For embedded systems, uClibc is clearly the winner.

3 Why NPTL for Embedded Systems?

uClibc supports multiple thread library models. What are the shortcomings of `linuxthreads`? Why is `nptl` better, or worse? The answer lies in the requirements, software and hardware, of the embedded platform being developed. We need to first compare `linuxthreads` and `nptl` to choose the thread library that best meets the needs of our platform. Table 2 lists the key features the two thread libraries have to offer.

Using the table above, the `nptl` model is useful in systems that do not have severe memory constraints, but need threads to respond quickly and efficiently. The `linuxthreads` model is useful mostly for resource constrained systems still needing basic thread support. As mentioned at the beginning of this paper, embedded systems with faster processors and greater memory resources are being required to do more, with less. Using NPTL in conjunction with the already small C library provided by uClibc, creates a perfectly balanced embedded Linux system that has size, speed and an high-performance multi-threading.

Feature	glibc	uClibc
LGPL	Y	Y
Complete POSIX compliance	Y	N
Binary compatibility across releases	Y	N
NSS Support	Y	N
NIS Support	Y	N
Locale support	Y	Y
Small disk storage footprint	N	Y
Small runtime memory footprint	N	Y
Supports MMU-less systems	N	Y
Highly configurable	N	Y
Simple build system	N	Y
Built-in configuration system	N	Y
Easily maintained	N	Y
NPTL Support	Y	Y
Linuxthreads Support	N (See below)	Y
Support many processor architectures	N (See below)	Y

Table 1: Library Feature Comparison

Feature	Description	LinuxThreads	NPTL
Storage Size	The actual amount of storage space consumed in the file system by the libraries.	Smallest	Largest
Memory Usage	The actual amount of RAM consumed at runtime by the thread library code and data. Includes both kernel and user space memory usage.	Smallest	Largest
Number of Threads	The maximum number of threads available in a process.	Hard Coded Value	Dynamic
Thread Efficiency	Rate at which threads are created, destroyed, managed, and run.	Slowest	Fastest
Per-Thread Signals	Signals are handled on a per-thread basis and not the process.	No	Yes
Inter-Thread Synchronization	Threads can share synchronization primitives like mutexes and semaphores.	No	Yes
POSIX.1 Compliance	Thread library is compliant	No	

Table 2: Thread Library Features

4 uClibc NPTL Implementation

The following sections outline the major technical components of the NPTL implementation for uClibc. For the most part, they should apply equally to glibc's implementation except where noted. References to additional papers and information are provided should the reader wish to delve deeper into the inner workings of various components.

4.1 TLS—The Foundation of NPTL

The first major component needed for NPTL on Linux systems is Thread Local Storage (TLS). Threads in a process share the same virtual address space. Usually, any static or global data declared in the process is visible to all threads within that process. TLS allows threads to have their own local static and global data. An excellent paper, written by Ulrich Drepper, covers the technical details for implementing TLS for the ELF binary format [6]. Supporting TLS required extensive changes to binutils [7], GCC [8] and glibc [1]. We cover the changes made in the C library necessary to support TLS data.

4.1.1 The Dynamic Loader

The dynamic loader, also affectionately known as `ld.so`, is responsible for the run-time linking of dynamically linked applications. The loader is the first piece of code to execute before the main function of the application is called. It is responsible for loading and mapping in all required shared objects for the application.

For non-TLS applications, the process of loading shared objects and running the application is trivial and straight-forward. TLS data types complicate dynamic linking substantially. The loader must detect any TLS sections, allocate initial memory blocks for any needed TLS data, perform initial TLS relocations, and later perform additional TLS symbol look-ups and relocations during the execution of the process. It must also deal with the loading of shared objects containing TLS data during program execution and properly allocate and relocate its data. The TLS paper [6] provides ample overview of how these mechanisms work. The document does not, however, currently cover the specifics of MIPS-specific TLS storage and implementation. MIPS TLS information is available from the Linux/MIPS website [9].

Adding TLS relocation support into uClibc's dynamic loader required close to 2200 lines of code to change. The only functionality not available in the uClibc loader is the handling of TLS variables in the dynamic loader itself. It should also be noted that statically linked binaries using TLS/NPTL are not currently supported by uClibc. Static binaries will not be supported until all processor architectures capable of supporting NPTL have working shared library support. In reality, shared library support of NPTL is a prerequisite for debugging static NPTL support. Thank you to Daniel Jacobowitz for pointing this out.

4.1.2 TLS Variables in uClibc

There are four TLS variables currently used in uClibc. The noticeable difference that can be observed in the source code is that they have an additional type modifier of `__thread`. Table 3 lists the TLS in detail. There are 15 additional TLS variables for locale support that were not ported from glibc. Multi-threaded locale support with NPTL is currently not supported with uClibc.

Variable	Description
errno	The number of the last error set by system calls and some functions in the library. Previously it was thread-safe, but still shared by threads in the same process. For NPTL, it is a TLS variable and thus each thread has its own instantiation.
h_errno	The error return value for network database operations. This variable was also previously thread safe. It is only available internally to uClibc.
_res	The resolver context state variable for host name look-ups.
RPC_VARS	Pointer to internal Remote Procedure Call (RPC) structure for multi-threaded applications.

Table 3: TLS Variables in uClibc

4.2 Futexes

Futexes [10] [11] are fast user space mutexes. They are an important part of the locking necessary for a responsive pthreads library implementation. They are supported by Linux 2.6 kernels and no code porting was necessary for them to be usable in uClibc other than adding prototypes in a header file. glibc also uses them extensively for the file I/O functions. Futexes were ported for use in uClibc's I/O functions as well, although not strictly required by NPTL. Futex I/O support is a configurable for uClibc and can be selected with the `UCLIBC_HAS_STDIO_FUTEXES` option.

4.3 Asynchronous Thread Cancellation

A POSIX compliant thread library contains the function `pthread_cancel`, which cancels the execution of a thread. Please see the official definition of this function at The Open Group website [12]. Cancellation of a thread must be done carefully and only at certain points in the library. There are close to 40 functions where thread cancellation must be checked for and possibly handled. Some of these are heavily used functions like `read`, `write`, `open`, `close`, `lseek` and others. Extensive changes were made to uClibc's C library core in order to support thread cancellation. A list of these are available on the NPTL uClibc development site [13].

4.4 Threads Library

The code in the nptl directory of glibc was initially copied verbatim from a snapshot dated 20050823. An entirely new set of build files were created in order to build NPTL within uClibc. Almost all of the original files remain with the exception of Asynchronous I/O (AIO) related code. All backwards binary compatibility code and functions have been removed. Any code that was surrounded with `#ifdef SHLIB_COMPAT` or associated with those code blocks was also removed. The uClibc NPTL implementation should be as small as possible and not constrained by old thread compatibility code. This also means that any files in the root `nptl` directory with the prefix of `old_pthread_` were also removed. Finally, there were minor header files changes and some functions renamed.

4.5 POSIX Timers

In addition to the core threads library, there were also code changes for POSIX Timers. These changes were integrated into the `librt` library in uClibc. These changes were not in the original statement of work from Broadcom, but were implemented for completeness. All the timer tests for POSIX timers associated with NPTL do pass, but were not required.

For those interested in further details of the NPTL design, please refer to Ulrich Drepper's design document [14].

5 uClibc NPTL Testing

There were a total of four test suites that the uClibc NPTL implementation was tested against. Hopefully, with all of these tests, uClibc NPTL functionality should be at or near the production quality of glibc's implementation.

5.1 uClibc Testsuite

uClibc has its own test suite distributed with the library source. While there are many tests verifying the inner workings of the library and the various subsystems, pthreads tests are minimal. There are only 7 of them, with another 5 for the dynamic loader. With the addition of TLS data and NPTL, these were simply not adequate for testing the new functionality. They were, however, useful as regression testing. The test suite can be retrieved with uClibc from the main site [2].

5.2 glibc Testsuite

The author would first like to convey immense appreciation to the glibc developers for creating such a comprehensive and usable test suite for TLS and NPTL. Had it not been for the tests distributed with their code, I would have released poor code that would have resulted in customer support nightmares. The tests were very well designed and extremely helpful in finding holes in the uClibc NPTL implementation. 182 selected NPTL tests and 15 TLS tests were taken from glibc and passed successfully with uClibc. There were a number of tests not applicable to uClibc's NPTL implementation that were omitted. For further details concerning the tests, please visit the uClibc NPTL project website [13].

5.3 Linux Test Project

The Linux Test Project (LTP) suite [15] is used to "validate the reliability, robustness, and stability of Linux." It has 2900+ tests that will not only test pthreads, but act as a large set of regression tests to make sure uClibc is still functioning properly as a whole.

5.4 Open POSIX Test Suite

To quote from the website [16], "The POSIX Test Suite is an open source test suite with the goal of performing conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation." This suite of tests is the most important indicator of how correct the uClibc NPTL implementation actually is. It tests pthreads, timers, asynchronous I/O, message queues and other POSIX related APIs.

5.5 Hardware Test Platform

All development and testing was done with an AMD Alchemy DBAu1500 board graciously

Source	Version
binutils	2.16.1
gcc	4.1.0
glibc	20050823
uClibc-nptl	20060318
Linux Kernel Headers	2.6.15
LTP	20050804
Open POSIX Test Suite (Distributed w/LTP)	20050804
buildroot	20060328
crosstool	0.38
Linux/MIPS Kernel	2.6.15

Table 4: Source and Tool Versions

donated by AMD. Broadcom also provided their own hardware, but it was not ready for use until later in the software development cycle.

The DBAu1500 development board is designed around a 400MHz 32-bit MIPS Au1500 processor core. The board has 64MB of 100MHz SDRAM and 32MB of AMD MirrorBit Flash memory. The Au1500 utilizes the MIPS32 Instruction Set and has a 16KB Instruction and 16KB Data Cache. (2) 10/100Mbit Ethernet Ports, USB host controller, PCI 2.2 compliant host controller, and other peripherals.

5.6 Software Versions

Table 4 lists the versions of all the sources used in the development and testing of uClibc NPTL. buildroot [17] was the build system used to create both the uClibc and glibc root filesystems necessary for running the test suites. crosstool [18] was used for building the glibc NPTL toolchain.

The actual Linux kernel version used on the AMD development board for testing is the released Linux/MIPS 2.6.15 kernel. The 2.6.16 release is not currently stable enough for testing and development. A number of system calls

glibc	uClibc
53m 8.983s	21m 33.129s

Table 5: Toolchain Build Times

appear to be broken along with serial text console responsiveness. The root filesystem was mounted over NFS.

6 uClibc NPTL Test Results

6.1 Toolchain Build Time

Embedded system targets do not usually have the processor and/or memory resources available to host a complete development environment (compiler, assembler, linker, etc). Usually development is done on an x86 host and the binaries are cross compiled for the target using a cross development toolchain. crosstool [18] was used to build the x86 hosted MIPS NPTL toolchain using glibc, and buildroot [17] was used to build the MIPS NPTL toolchain using uClibc.

Building a cross development toolchain is a time consuming process. Not only are they difficult to get working properly, they also take a long time to build. glibc itself usually must be built twice in order to get internal paths and library dependencies to be correct. uClibc on the other hand, need only be built once due to its simpler design and reduced complexity of the build system. The toolchains were built and hosted on a dual 248 Opteron system with 1GB of RAM, Ultra 160 SCSI and SATA hard drives. Times include the actual extraction of source from the tarballs for toolchain components. See Table 5. The toolchains above compile both C and C++ code for the MIPS target processor. Not only are uClibc's libraries

smaller (as you will see shortly), but creating a development environment is much simpler and less time consuming.

6.2 Library Code Size

Throughout our discussion, we have stressed the small size of uClibc as compared to glibc. Tables 6 and 7 show these comparisons of libraries and shared objects as compared to one another.

The size difference between the C libraries is dramatic. uClibc is better than 2 times smaller than glibc. uClibc's `libdl.a` is larger because some TLS functions only used for shared objects are being included in the static library. This is due to a problem in the uClibc build system that will be addressed. The `libnsl.a` and `libresolv.a` libraries are dramatically smaller for uClibc only because they are stub libraries. The functions usually present in the corresponding glibc libraries are contained inside uClibc. Finally, `libm.a` for uClibc is much smaller do to reduced math functionality in uClibc as compared to glibc. Most functions for handling `double` data types are not present for uClibc.

The shared objects of most interest are `libc.so`, `ld.so`, and `libpthread.so`. uClibc's main C library is over **2 times smaller** than glibc. The dynamic loader for uClibc is **4 times smaller**. glibc's dynamic loader is complex and larger, but it has to be in order to handle binary backwards compatibility. Additionally, uClibc's dynamic loader is cannot be executed as an application like glibc's. Although the NPTL pthread library code was ported almost verbatim from glibc, uClibc's library is **30% smaller**. Why? The first reason is that any backward binary compatibility code was removed. Secondly, the comments in the nptl directory of glibc say that the NPTL code should be compiled with the `-O2` compiler option. For uClibc, the `-Os` option was used to reduced the code size and yet NPTL still functioned perfectly. This optimization worked for MIPS, but other architectures may not be able to use this optimization.

6.3 glibc NPTL Library Test Results

The developers of NPTL for glibc created a large and comprehensive test suite for testing its functionality. 182 tests were taken from glibc and tested with uClibc's TLS and NPTL implementation. All of these tests passed with uClibc NPTL. For a detailed overview of the selected tests, please visit the uClibc NPTL project website.

6.4 Linux Test Project (LTP) Results

Out of over 2900+ tests executed, there were only 31 failed tests by uClibc. glibc also failed 31 tests. A number of tests that passed with uClibc, failed with glibc. The converse was also true. These differences will be examined at a later date. However, passing all the tests in the LTP is a goal for uClibc. Detailed test logs can be obtained from the uClibc NPTL project website.

6.5 Open POSIX Testsuite Results

Table 8 shows the results of the test runs for both libraries.

The first discrepancy observed is the total number of tests. glibc has a larger number of tests available because of Asynchronous I/O support and the `sigqueue` function, which is not currently available in uClibc. Had these features been present in uClibc, the totals would have

Static Library	glibc [bytes]	uClibc [bytes]
libc.a	3 426 208	1 713 134
libcrypt.a	29 154	15 630
libdl.a	10 670	36 020
libm.a	956 272	248 598
libnsl.a	161 558	1 100
libpthread.a	281 502	250 852
libpthread_nonshared.a	1 404	1 288
libresolv.a	111 340	1 108
librt.a	79 368	29 406
libutil.a	11 464	9 188
TOTAL	5 068 940	2 306 324

Table 6: Static Library Sizes (glibc vs. uClibc)

Shared Object	glibc [bytes]	uClibc [bytes]
libc.so	1 673 805	717 176
ld.so	148 652	35 856
libcrypt.so	28 748	13 676
libdl.so	16 303	13 716
libm.so	563 876	80 040
libnsl.so	108 321	5 032
libpthread.so	120 825	97 189
libresolv.so	88 470	5 036
librt.so	45 042	14 468
libutil.so	13 432	9 320
TOTAL	2 807 474	991 509

Table 7: Shared Object Sizes (glibc vs. uClibc)

RESULT	glibc	uClibc
TOTAL	1830	1648
PASSED	1447	1373
FAILED	111	83
UNRESOLVED	151	95
UNSUPPORTED	22	29
UNTESTED	92	60
INTERRUPTED	0	0
HUNG	1	3
SEGV	5	5
OTHERS	1	0

Table 8: OPT Results (glibc vs. uClibc)

most likely been the same. The remaining results are still being analyzed and will be presented at the Ottawa Linux Symposium in July, 2006. The complete test logs are available from the uClibc NPTL project website.

7 Conclusions

NPTL support in uClibc is now a reality. A fully POSIX compliant threads library in uClibc is a great technology enabler for embedded systems developers who need fast multithreading capability in a small memory footprint. The results from the Open POSIX Testsuite need to be analyzed in greater detail in order to better quantify what, if any POSIX support is missing.

8 Future Work

There is still much work to be done for the uClibc NPTL implementation. Below is a list of the important items:

- Sync NPTL code in uClibc tree with latest glibc mainline code.
- Implement NPTL for other processor architectures.
- Get static libraries working for NPTL.
- Merge uClibc-NPTL branch with uClibc trunk.
- Implement POSIX message queues.
- Implement Asynchronous I/O.
- Implement `sigqueue` call.
- Fix outstanding LTP and Open POSIX Test failures.

9 Acknowledgements

I would first like to acknowledge and thank God for getting me through this project. It has been a 9-1/2 month journey full of difficulty and frustration at times. His strength kept me going. Secondly, my wife Jennifer who was a constant encourager and supporter of me. I spent many weekends and evenings working while she kept the household from falling apart. Obviously, I would like to thank Broadcom Corporation for supporting this development effort and providing the code back to the community. Thanks to Erik Andersen from Code Poet Consulting who was my partner in this endeavor, handling all the contracts and legal issues for me. Thank you to AMD for supplying me multiple MIPS development boards free of charge. Special thanks to Mathieu Chouinard for formatting my paper in the final hours before submittal. Finally, I would like to dedicate this paper and entire effort to my first child, Zachary James Hill who just turned a year old on March 5th, 2006. I love you son.

References

[1] GNU C Libary at http://www.gnu.org/software/libc/ http://sourceware.org/glibc/

[2] uClibc at http://www.uclibc.org/

[3] Newlib at http://sourceware.org/newlib/

[4] Diet Libc at http://www.fefe.de/dietlibc/

[5] Klibc at ftp://ftp.kernel.org/pub/linux/libs/klibc/

[6] ELF Handling for Thread Local Storage at http://people.redhat.com/drepper/tls.pdf

[7] Binutils at http://www.gnu.org/software/binutils/

[8] GCC at http://gcc.gnu.org/

[9] NPTL Linux/MIPS at http://www.linux-mips.org/wiki/NPTL

[10] Hubertus Franke, Matthew Kirkwood, Rusty Russell. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, pages 479–494, June 2002.

[11] Futexes Are Tricky at http://people.redhat.com/drepper/futex.pdf

[12] pthread_cancel function definition at http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_cancel.html

[13] uClibc NPTL Project at http://www.realitydiluted.com/nptl-uclibc/

[14] The Native POSIX Thread Library for Linux at http://people.redhat.com/drepper/nptl-design.pdf

[15] Linux Test Project at http://ltp.sourceforge.net/

[16] Open POSIX Test Suite at http://posixtest.sourceforge.net/

[17] buildroot at http://buildroot.uclibc.org/

[18] crosstool at http://www.kegel.com/crosstool/