Pthreads Profiler

by
Ryan Van Quinlan

Computer Science Department
College of Engineering
California Polytechnic State University
2015

Date Submitted: December 8, 2015
Advisor: Dr. Phillip Nico
TABLE OF CONTENTS

PAGE

ABSTRACT ........................................................................................................ ii

LIST OF FIGURES ........................................................................................ iii

LIST OF TABLES .......................................................................................... iv

Section

I. Introduction................................................................................................. 1
II. Background ............................................................................................... 3
III. Description ................................................................................................. 8
IV. Evaluation .................................................................................................. 13
V. Conclusions ............................................................................................... 19

Appendix

A. Source Code .............................................................................................. 20
B. Relevant Man Pages ................................................................................... 29
C. References ................................................................................................. 30
The goal of this senior project was to design and implement a program which would allow users to profile programs that utilize the Pthreads library. This profiler allows users to measure load balance and lock contention between threads as well as discover deadlocks. These tools can be essential to developers of multi-threaded applications, which are notoriously difficult to debug. Additionally, this tool can help find vital performance benefits by providing relative execution measurements. The profiler is written in C++ and utilizes the standard template library as well as glibc extensions.
LIST OF FIGURES

FIGURE

I. Processor Frequency Scaling Over Time .................................................. 6
II. History of transistor count, core count, frequency, and power ............... 7
III. Contention Sample Output ................................................................. 15
IV. Deadlock Sample Output ................................................................. 17
LIST OF TABLES

TABLE

I. two_threads_unbalanced sample run results ........................................ 13
Introduction

As Herb Sutter wrote, “The Free Lunch Is Over.”[1] Today’s software developers are being challenged by hardware developers to write code that can take advantage of multiple cores in order to continue maintaining Moore’s Law. As cycle speeds plateau and core counts explode, software can no longer rely on next year’s hardware for automatic performance gains without having to make changes to the underlying code. As the burden shifts we are beginning to see new technologies emerge such as CUDA, OpenCL, Go, and Rust.[2,3,4,5] However, many developers can not wait for new technologies to mature or rewrite millions of lines of code in a new language. Developers are looking for immediate solutions that can be applied to the decades of accrued codebases currently in existence.

Finding and implementing these solutions has proven difficult. Not only due to implementation struggles, but debugging and result verification as well. Some of the most difficult challenges arise from attempting to balance workloads, discover deadlocks, and manage lock contention. My project addresses these issues by allowing developers to profile programs that utilize the Pthreads library without any modification to their existing code. Deadlocks are automatically detected during runtime while also optionally providing a stack trace of where in the program the deadlock was found. Lock contention measurements tell users how many times a
mutex was contested by threads as well as the total time a mutex was held during program execution. Load balance reports show users the relative measurements of how long each thread was working. While none of these measurements are completely precise, the insights they provide can allow developers to better take advantage of underlying hardware as well as discover and remedy the unacceptable problem of a deadlock. Future improvements will focus more on improving robustness over additional features, as this project prioritizes a reliable narrow scope.
Background

Before Pthreads, developers would often design and implement their own thread libraries. The methods and specifications could vary substantially, inhibiting code reuse, readability, and ease of adoption for users. Pthreads (POSIX threads) is a thread library interface specified by the IEEE POSIX 1003.1[6] standard. The creation of this standard API greatly facilitates the use of threads across platforms and hardware. Some of the most integral thread operations are covered by the Pthreads specification, such as management, synchronization and conditions. As a result, writing cross-platform, POSIX compliant, multi-threaded applications has become much easier for developers.

However, even with the availability of standardized libraries, the difficulties of working with threads still varies between programs. One of the first obstacles a developer can face is how to best partition a problem between threads; otherwise known as load balancing. Predicting how much work a thread will actually have to perform while considering other factors such as hardware access and the locality of data in time and space (which affects cache misses and coherency) makes solving this problem practically impossible without actual measurements during execution. Managing load balance not only varies between programs, but as programs evolve it can vary between different versions of the same program as well. Fortunately, a
profiling tool can help guide developers as changes are made. My project measures load balance by keeping track of which threads are working at specific time intervals and reports how often each thread was busy. This provides the developer an estimation of how much relative time each thread spent working. By profiling programs as they evolve and measuring the impact of modifications on each thread, developers can use profiling results as a guide on how to best modify work distribution amongst threads. However, there are some issues to be aware of. When measuring load balance, there is a delicate balance between accuracy and profiler impact. If a tool attempts to take measurements too frequently it would ironically actually be measuring the imbalance it caused. On the other hand, if measurements are not made often enough, there will not be enough data points to make worthwhile judgments. Fortunately, this issue is manageable as any decent attempt at balance is still likely to highlight performance outliers which are where most gains can be found.

Another obstacle comes from managing the shared data between threads. Developers must sometimes lock critical regions of code in order to guarantee sequential execution and prevent possible race conditions. Large codebases can often accumulate a large number of critical regions and locks. The performance gains acquired by using threads are hindered as the size and number of critical regions grows. Therefore, in order to take full advantage of thread performance, a
developer must know which locks are most inhibiting the executing threads. That is, which locks are the most contended between threads, and which locks are held for the longest time. Measuring lock contention allows developers to effectively spend their time and effort on regions that are likely to offer the largest performance advantages. Like load balance, lock contention can be difficult to estimate without runtime measurements and when discussing the most valuable resource available, the developer's time, any tool that can improve accuracy is invaluable.

Finally, a third obstacle multi-threaded developers can face is deadlock. Deadlock occurrences are often defined by the Coffman conditions being met simultaneously: mutual exclusion, hold and wait, no preemption, and circular wait. Pthread mutexes guarantee the first and third conditions. That is, a mutex can only be held by one thread at a time and a thread can not be forced to release a mutex, it must voluntarily release it. The second and fourth conditions can occur when a developer overlooks the order of lock acquisition, possible execution paths, or makes some other logical error. Assuming these conditions are met, the first step to handling a deadlock is detection. Unfortunately, statically detecting deadlocks would be equivalent to solving the halting problem, which means deadlocks can only be accurately detected at runtime. Even then, runtime deadlocks can be hard for users to distinguish from very poor performance, thrashing, or infinite loops. Another difficulty is that deadlock occurrences sometimes depend on a race
condition or specific execution path which means it may not occur during every run or even during most runs. Finally, assuming the deadlock has been detected at runtime, recovering from a deadlock would depend on a program’s current state meaning in general deadlocks are fatal. These issues make deadlocks potentially catastrophic, particularly in critical systems such as health and finance. While the only true deadlock countermeasures are careful developers, my project attempts to help alleviate this burden with runtime detection and providing as much information as possible to help fix it.

Figure I: Processor Frequency Scaling Over Time

https://queue.acm.org/detail.cfm?id=2181798
Figure II: History of transistor count, core count, frequency, and power
http://rsta.royalsocietypublishing.org/content/372/2022/20130319

Despite all the difficulties of multi-threaded development, the benefits still make it worthwhile. Figures I and II illustrate processor trends and the need to utilize multiple cores. Notably, processor clock speeds have stagnated, and an increasing number of cores is being substituted by hardware manufacturers to continue meeting performance demands. Software must adapt to utilize the increasing number of cores as well in order to avoid stagnation. Multi-threaded development is a relatively new frontier with new challenges to overcome and design considerations to be made. Quality tools developers can equip themselves with will help ease the software world into these new areas.
Description

The Pthreads profiler is easy to use with an existing client program. For example, it can simply be invoked with `./rain ./client_program` from the terminal.

The rain program is a small script that sets the LD_LIBRARY_PATH and LD_PRELOAD environment variables. I add the current directory to LD_LIBRARY_PATH so that it will be searched for shared libraries and add the profilers shared object file to LD_PRELOAD to have it loaded before the actual Pthreads library. The profilers shared object file defines a number of functions with identical signatures to the actual Pthreads library so when the user calls a Pthreads function the profiler’s functions will actually be invoked. This is how the profiler hooks into the client program and performs various bookkeeping before continuing to invoke the real Pthreads library on behalf of the client.

In order to actually invoke the real Pthreads library functions instead of recursively calling the profiler’s functions, the profiler invokes dlsym() to find the next function with that name on the dynamic linker’s stack. I encountered an odd quirk of C++ doing this. C++ does not support casting void pointers (what dlsym() returns) to function pointers directly (it is optionally supported in c++0x). To get around this, I assert the pointer size and use a reinterpret_cast. Additionally, as I found myself writing this wrapper code often, I put the code in a macro to simplify
wrapping around a Pthreads function.

To initialize the real Pthreads library functions I had to first check if the real functions were initialized at the beginning of each of the profiler’s wrapper functions (because I could not know which the client would invoke first) and initialize them if not. In order to do this check, I needed thread safety without having access to any of the Pthreads library. I had originally planned on doing this project in C, however, this is where I switched to C++ for atomics support. C has support for atomics through stdatomics.h, however, I discovered that GCC 4.8 (the version I was using at the time) had a bug which made atomics unavailable. GCC 4.9 fixed this but was not made available for my distro until last summer and I decided that I was too far along with C++ to switch back to plain C.

Once I could be confident the real Pthreads library functions had been loaded, I next had to initialize my bookkeeping data structures. I needed to use Pthreads mutexes to manage my data structures because the profiler’s wrapper functions needed to remain thread safe. In the pthread_create() wrapper, I check if any threads already exist, and if not, I perform initialization and setup signal timers (currently set for 100Hz). During this stage of development, I also implemented a version with a dedicated thread to handle performing measurements at the set time intervals. While I found the dedicated thread version to more accurately match expected wall clock time a thread spends working, I decided in the end that the
thread version would be much more complicated to implement and potentially have more impact on the client program.

Signals on Linux are process specific, not thread specific. When a signal was caught there was no way of knowing which thread in the process would handle it. Therefore, I had to iterate over every thread, and send a second signal to each thread individually (including whichever thread had handled the original signal). I found pthread_kill() could handle this task; however, pthread_sigqueue, a glibc specific function, allowed me to pass a variable along with the signal as well. I used this extra data to send the thread index, which was invaluable to me because signals can not use mutexes which meant all my data access within the signal handler had to be careful about race conditions. By passing an index, I could be sure each thread was only modifying its particular portion of an array in the handler. Now that each thread was handling its own signal, my challenge was deciding whether or not a thread had performed work since the last signal. To do this, I decided to read each thread’s register set and check if it had changed since the last signal. This has so far proven to be a rather effective method, especially given the instruction pointer.

With everything else in place, lock contention was much simpler to implement. Using Pthread mutexes to protect data access and the C++ standard library, it was relatively easy to keep track of how many times a lock was contested and how long it was held. The real key here was pthread_mutex_trylock, which
failed if the lock was already held by another thread and meant I could increment the contested counter. Similarly, keeping track of how long a lock was held was simply a matter of checking the time during pthread_mutex_lock() and pthread_mutex_unlock().

Deadlock detection was trickier. First I had to figure out an algorithm to actually detect deadlocks. After doing some research, I discovered people typically implement this using a dependency graph. However, I thought that solution seemed more resource intensive and complicated than required. Instead, I wrote an algorithm which simply uses two tables: one to keep track of who currently owns locks, and another to keep track of which lock each thread is currently waiting to acquire. Using these two small tables, I wrote a small recursive function to check if the given thread acquiring the requested lock would cause a deadlock. Additionally, I wanted to provide the user with a stack trace if a deadlock occurred so that the profiler could be more helpful than simply reporting that a problem had occurred. To provide the trace, I again utilized glibc which provides the convenient backtrace() and backtrace_symbols() functions. However, there are two issues with this approach. First, the functions require the client program to be compiled with the -rdynamic flag. This is unfortunate because it may not be a possibility for everyone. The second issue was that these functions use Pthreads mutexes, which is where I had just detected the deadlock. Therefore, my only option was to create a
flag to disable my profiler’s functions that wrapped around the Pthreads mutex function calls so that backtrace could simply invoke the original Pthreads functions without causing infinite recursion or affecting the client’s results.
Evaluation

To evaluate the various features of this program, several test cases were written. Most of the tests focus on one specific area of the profiler's capabilities, while others are instead more general attempts which might be found in real world applications. A few of these tests will be discussed here as a measure of the program's correctness.

two_threads_unbalanced creates two threads that each busy wait until a given time has elapsed. The program takes two arguments which specify how long each thread should “work”. Table I shows the result of various sample executions.

<table>
<thead>
<tr>
<th>Run Number</th>
<th>Thread Number</th>
<th>Real Time Worked (s)</th>
<th>Reported Sigcounts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>164</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>10</td>
<td>1060</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>669</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>524</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>641</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>86</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>7</td>
<td>1287</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td>1285</td>
</tr>
</tbody>
</table>

Table I: two_threads_unbalanced sample run results
For each run, the table shows the real time each thread spent working in seconds (these times are provided as arguments to the program), as well as the reported number of signals received by the profiler which found that the thread had performed work. As illustrated, the program gives relative measurements of load balance between threads. The reported sigcounts after the first run clearly shows that thread 1 performs much more work compared to thread 0. Runs 2 and 3 also identify room for improvement in load balance for the user. However, these runs also illustrate another important point: the reported sigcounts are useful only as relative measurements to other threads' sigcounts. That is, they do not accurately represent time spent working or amount of work done. This was never the intended outcome. Rather, the reported sigcounts meet their goal of reporting relative load balance measurements to users.

A sample contention test program simply has a single shared array of sums which multiple threads attempt to lock, modify, and release. The given test creates 10 threads, which each increment 10 elements in an array 100 times. By placing separate locks around the modification of each element, as well as sleeping a short time after each modification, we can be relatively confident in a fair amount of contention for each lock. Table III shows the output of a sample run of the contention test program described.
The first part of the output in Figure III again shows the load balance measurements described earlier. After the sigcounts, the user is told how many mutexes the profiler found to be used in total during execution. The table shows the address of each mutex, how many times each was locked, how many times each was contended (that is, how many times another thread attempted to lock the mutex while it was already held), and the total time it was locked during execution in milliseconds. The mutex address can be used by the user to find the mutex to which each result corresponds. The locked and contention reports are what might interest the user the most. Here we can see if any locks are frequently contented, and perhaps adjust the code accordingly to try and reduce the reported contention because, as explained
previously, when a thread must wait on a lock this can often waste time and be a major source of performance loss. Finally, we see that the locks still worked as expected and the resulting data array values are what we would expect.

The deadlock detection feature was tested with several programs. The programs tested simple deadlock using just two threads and sleeping between locks, more complex deadlocks between multiple threads and a chain of poorly ordered locks and sleeps, a thread deadlocking with itself by trying to acquire the same lock twice, and finally a test with no deadlock as a control. These tests are all passed with the profiler. However, there is a very slight chance of false positives which never occurred during testing. The core reason for this is that the profiler can not place locks around the locking and unlocking without causing a deadlock. In other words, there is a slight chance at a race condition between the actual locking/unlocking and the deadlock test. This could result in two possible incorrect outcomes. The first outcome is the profiler reports a deadlock which does not actually occur and the second is the profiler does not report a deadlock which does occur. The first outcome is not very bad at all. If this happens, the deadlock was actually possible and just managed to not happen this time but must still be addressed by the user. The second outcome is much worse, as this means there is a slight chance that the profiler's deadlock detection has failed. Therefore, this profiler can not be relied upon 100% and the user must be aware of this potential for failure. Nonetheless, the tool can be very helpful to users and is reliable the vast
majority of the time. Figure IV shows sample output of a detected deadlock.

Figure IV: Deadlock Sample Output

The first part of the sample output is from the test program. The test expects that when it attempts to lock the mutex at address 0x6020c0 in the thread with address 06020a8 a deadlock will occur. The profiler does indeed detect the deadlock as expected and provides a stack trace to the user. Although it can be difficult to read, hidden in the stack trace is a function named workTwo in the deadlock program which is where the thread attempts to lock the mutex which causes the deadlock.

The largest detected shortcoming of the profiler is recursive creation and joining of threads. There still seems to be a bug in the profiler that will cause segmentation faults in programs that recursively create and join threads. This seems to be a difficult race condition as it does not always occur and the odds increase as the depth of recursion grows. This was first detected while attempting to profile an implementation of mergesort which utilized Pthreads. The test was then further simplified to a Fibonacci implementation in order to attempt to narrow down the
issue. When attempting to calculate Fibonacci numbers below around 6, the problem rarely occurs, around 7 and 8 it seems to occur about half of the time and above that it occurs frequently. This bug still requires further investigation and work.
Conclusions

For the most part, the Pthread profiler program meets its main three goals of providing load balance and lock contention measurements to the user and detecting deadlocks. However, there are still remaining issues. One of the core issues is the known remaining bug of a crash while attempting to profile recursive thread creation and joining. Also, the possibility of failure during deadlock detection is unfortunate, however, a solution may not be possible without running the program in kernel space and a great deal of workaround. Aside from bugs, future work and features may expand the scope of the program. A custom means of acquiring a stack trace would be ideal so as to eliminate the -rdynamic flag requirement from client program compilation. Additionally, the profiler currently covers only a small subset of the full Pthreads library. Therefore, it is currently only useful for programs which use some of the more core functions such as creating/joining threads and locking/unlocking mutexes. As the scope of coverage is increased, the scope of programs able to be accurately profiled would also increase. Finally, long term goals may include the support of additional platforms such as Windows or Mac.
/********* librain.cpp *********/
#include <pthread.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <sys/time.h>
#include <sys/syscall.h>
#include <execinfo.h>
#include <dlfcn.h>
#include <ucontext.h>
#include <errno.h>
#include <string>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <atomic>   // pthreads has no atomic support, so this is
// used for initializing pthreads library functions

typedef struct RegSet {
   // use signal context to read register set
   greg_t generalRegister[NGREG];
} RegSet;

// how many threads are actually running
static unsigned int activeThreadCount = 0;

// total threads that ran at once (max active)
static unsigned int totalThreadCount = 0;

// access to client's threads
static std::vector<pthread_t *> clientThreads;

// count of signals thread receives
static std::vector<sig_atomic_t> sigCounts;

// set of registers for each thread at last profile interval
static std::vector<RegSet> lastRegSet;

// data structures maintained for deadlock detection
// map of mutexes to the thread id holding the mutex lock
static std::unordered_map<pthread_mutex_t *, int> lockHolders;

// map of thread ids to the mutex it is waiting to acquire (if any)
static std::unordered_map<int, pthread_mutex_t *> threadWaiters;
typedef struct MutexData {
    int lockCount;
    int contention;

    uint64_t lockTimeStamp;
    uint64_t lockTimeTotal;
} MutexData;

// map mutexes to count of times they were already held when
// another thread tried to acquire them
static std::unordered_map<pthread_mutex_t *, MutexData> mutexData;

// per-thread flag to disable some aspects of profiler
static __thread bool disableRain = false;

// pthread functions initialized?
static std::atomic_flag real_pthread_initialized = ATOMIC_FLAG_INIT;
static bool function_wrappers_initialized = false;

// lock for internal operations
static pthread_mutex_t rain_lock = PTHREAD_MUTEX_INITIALIZER;

// need to be able to call the actual pthread library functions
static int (*real_pthread_create)(pthread_t*, const pthread_attr_t*,
    void (*)(void*), void*) = NULL;
static int (*real_pthread_join)(pthread_t, void **) = NULL;
static void (*real_pthread_exit)(void *)__attribute__((noreturn)) = NULL;
static int (*real_pthread_mutex_lock)(pthread_mutex_t*) = NULL;
static int (*real_pthread_mutex_unlock)(pthread_mutex_t*) = NULL;

#define WRAP_FUNCTION(FUN_NAME)\
    do {\
    /* c++ doesn't support casting void pointers to function pointers,
workaround:*/\
    /* http://stackoverflow.com/questions/1096341/function-pointers-casting-in-
c*/\
        static_assert(sizeof(void*) == sizeof(real_##FUN_NAME), "pointer cast impossible");\
        *reinterpret_cast<void**>(&real_##FUN_NAME) = dlSym(RTLD_NEXT, #FUN_NAME);\
        if (real_##FUN_NAME == NULL) {\
            fprintf(stderr, "ERROR: RAIN: #FUN_NAME, dlSym: %s\n", dlerror());\
        }\
    } while (0)

static uint64_t timeNow() {
    struct timespec t;
    int res = clock_gettime(CLOCK_MONOTONIC, &t);
    if (res) {
fprintf(stderr, "ERROR: RAIN: clock_gettime, %d\n", res);
    return 0;
}
return t.tv_sec * 1000000000 + t.tv_nsec;
}

static void printBacktrace() {
    /* glibc specific backtrace functions for stack traces
    * backtrace_symbols requires client programs to be compiled with -rdynamic
    * however, if they aren't it just won't give function names and
    * still works gracefully
    */

    /* backtrace uses a mutex, so we need to disable some wrappers temporarily */
    disableRain = true;

    /* gnu.org: 200 possible entries should probably cover all programs */
    static const int NUM_RET_ADDR = 200;
    void *callstack[NUM_RET_ADDR];
    int i, frames = backtrace(callstack, NUM_RET_ADDR);
    char **strs = backtrace_symbols(callstack, frames);
    for (i = 0; i < frames; ++i) {
        printf("%s\n", strs[i]);
    }
    disableRain = false;
}

static void init_real_pthreads() {
    // initialize real pthread function pointers
    WRAP_FUNCTION(pthread_create);
    WRAP_FUNCTION(pthread_join);
    WRAP_FUNCTION(pthread_exit);
    WRAP_FUNCTION(pthread_mutex_lock);
    WRAP_FUNCTION(pthread_mutex_unlock);
    function_wrappers_initialized = true;
}

static void sigprof_handler(int sig_nr, siginfo_t* info, void *context) {
    (void)sig_nr;
    (void)info;
    (void)context;
    /*
    // to block SIGPROF while handling it:
    sigset_t block_set;
    sigemptyset(&block_set);
    sigaddset(&block_set, SIGPROF);
    pthread_sigmask(SIG_BLOCK, &block_set, NULL);
    //sigprocmask(SIG_BLOCK, &block_set, NULL);
    */
    unsigned int t;
    for (t = 0; t < totalThreadCount; ++t) {
        sigval sval;
        sval.sival_int = t;
// send SIGUSR1 to each thread so they can all grab a call stack
if (clientThreads[t]) {
    // pthread_sigqueue is GNU specific, but allows
    // passing an integer value to the signal handler
    int res = pthread_sigqueue(*clientThreads[t], SIGUSR1, sval);
    if (res) {
        // can be used for extra debugging, however can get flooded with
        // this error because it takes a few
        // instructions between the actual calls to
        // pthread_create/pthread_join and updating my data structures
        fprintf(stderr, "ERROR: RAIN: sigprof_handler: pthread_sigqueue:
            \ SIGUSR1 signal not sent: %d, %d\n", t, res);
    }
    pthread_kill(*clientThreads[t], SIGUSR1);
}
//pthread_sigmask(SIG_UNBLOCK, &block_set, NULL);

static void sigusr1_handler(int sig_nr, siginfo_t* info, void *context) {
    (void) sig_nr;
    unsigned int t = info->si_value.sival_int;

    // instead of just counting signals received by a thread, also check
    // current register set to see if it has changed from last check
    // (so we can get some idea of whether or not thread has actually done work)
    bool regChanged = false;
    ucontext_t *ucontext = (ucontext_t*)context;
    for (int i = 0; i < NGREG; i++) {
        if (ucontext->uc_mcontext.gregs[i] != lastRegSet[t].generalRegister[i]) {
            regChanged = true;
        }
    }
    lastRegSet[t].generalRegister[i] = ucontext->uc_mcontext.gregs[i];
    if (regChanged) {
        ++sigCounts[t];
    }
}

// called just before first thread is created
static void begin() {
    // gcc 4.9 has a bug with unordered_map that causes a floating point exception
    // reserve here as a workaround to prevent it
    // https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61143
    lockHolders.reserve(1);
    threadWaiters.reserve(1);
    mutexData.reserve(1);

    totalThreadCount = 0;
    clientThreads.clear();
    sigCounts.clear();
    lastRegSet.clear();

    struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_sigaction = sigprof_handler;
sa.sa_flags = SA_RESTART | SA_SIGINFO;
sigemptyset(&sa.sa_mask);
sigaction(SIGPROF, &sa, NULL);

struct sigaction sa2;
memset(&sa2, 0, sizeof(sa2));
sa2.sa_sigaction = sigusr1_handler;
sa2.sa_flags = SA_RESTART | SA_SIGINFO;
sigemptyset(&sa2.sa_mask);
sigaction(SIGUSR1, &sa2, NULL);

static struct itimerval _RAIN_timer;
_RAIN_timer.it_interval.tv_sec = 0;
_RAIN_timer.it_interval.tv_usec = 1000000 / 100; /* 100hz */
_RAIN_timer.it_value = _RAIN_timer.it_interval;
if (setitimer(ITIMER_PROF, &_RAIN_timer, NULL)) {
    fprintf(stderr, "ERROR: RAIN: begin: timer could not be initialized: %s
", strerror(errno));
}

// called just after last thread joined
static void finish() {
    struct itimerval _RAIN_timer = {0};
    if (setitimer(ITIMER_PROF, &RAIN_timer, NULL)) {
        fprintf(stderr, "ERROR: RAIN: finish: timer could not be stopped: %s
", 
                strerror(errno));
    }

    for (unsigned int t = 0; t < sigCounts.size(); t++) {
        printf("thread %d sigcount %d\n", t, sigCounts[t]);
    }

    if (mutexData.size()) {
        printf("%lu mutexes used\n", mutexData.size());
        printf("Mutex\tLocked\tContention\tTotal Time (ms)\n");
        for (auto kv : mutexData) {
            printf("%p\t%d\t%d\t%.4f\n", kv.first, kv.second.lockCount, 
                 kv.second.contention, kv.second.lockTimeTotal / 1000000.0);
        }
    }
}

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg) {

    if (!real_pthread_initialized.test_and_set()) {
        init_real_pthreads();
    }

    while (!function_wrappers_initialized);
    // lock this all up in case of threads creating threads
    int res = real_pthread_mutex_lock(&rain_lock);
    if (res) {

        usleep(1000000 / 100); /* 100hz */
        _RAIN_timer.it_value = _RAIN_timer.it_interval;
        if (setitimer(ITIMER_PROF, &RAIN_timer, NULL)) {
            fprintf(stderr, "ERROR: RAIN: begin: timer could not be initialized: %s
", 
                    strerror(errno));
        }

        for (unsigned int t = 0; t < sigCounts.size(); t++) {
            printf("thread %d sigcount %d\n", t, sigCounts[t]);
        }

        if (mutexData.size()) {
            printf("%lu mutexes used\n", mutexData.size());
            printf("Mutex\tLocked\tContention\tTotal Time (ms)\n");
            for (auto kv : mutexData) {
                printf("%p\t%d\t%d\t%.4f\n", kv.first, kv.second.lockCount, 
                     kv.second.contention, kv.second.lockTimeTotal / 1000000.0);
            }
        }
    }

    return res;
}
if (activeThreadCount++ == 0) {
    begin(); // first thread, initialize
}
int ret = real_pthread_create(thread, attr, start_routine, arg);
clientThreads.push_back(thread);

RegSet initRegSet;
alastRegSet.push_back(initRegSet);
sigCounts.push_back(0);
++totalThreadCount;

int pthread_join(pthread_t thread, void **value_ptr) {
    if (!real_pthread_initialized.test_and_set()) {
        init_real_pthreads();
    }
    while (!function_wrappers_initialized);
    int ret = real_pthread_join(thread, value_ptr);
    int res = real_pthread_mutex_lock(&rain_lock);
    if (res) {
        // if unable to lock, just try not to disrupt host program
        fprintf(stderr, "ERROR: RAIN: pthread_join: could not lock thread: %d\n", res);
        return ret;
    }
    unsigned int t;
    for (t = 0; t < totalThreadCount; ++t) {
        if (clientThreads[t] && pthread_equal(*clientThreads[t], thread)) {
            clientThreads[t] = 0;
            break; // found the thread
        }
    }
    if (--activeThreadCount == 0) {
        finish(); // last thread, cleanup, output traces
    }
    res = real_pthread_mutex_unlock(&rain_lock);
    if (res) {
        // likely means rain_lock does not own the mutex somehow
    }
    return ret;
}
fprintf(stderr, "ERROR: RAIN: pthread_join: could not unlock thread: %d\n", res);
    return ret;
}

void pthread_exit(void *value_ptr) {
    if (!real_pthread_initialized.test_and_set()) {
        init_real_pthreads();
    }
    while (!function_wrappers_initialized);
    real_pthread_exit(value_ptr);
}

static bool deadlockDetectRecur(int threadRequesting, pthread_mutex_t *mutex) {
    if (lockHolders.count(mutex) == 0) {
        return 0;   // no thread currently holding this lock
    }
    int holder = lockHolders[mutex];
    if (holder < 0) {
        return 0;   // no thread currently holding this lock
    }
    if (holder == threadRequesting) {
        return 1;   // cycle detected
    }
    if (threadWaiters.count(holder) == 0) {
        return 0;   // holder is not waiting on any locks
    }
    return deadlockDetectRecur(threadRequesting, threadWaiters[holder]);
}

// returns 1 if given thread trying to acquire given mutex creates a deadlock, else 0
static bool deadlockDetect(int thread, pthread_mutex_t *mutex) {
    return deadlockDetectRecur(thread, mutex);
}

int pthread_mutex_lock(pthread_mutex_t *mutex) {
    if (!real_pthread_initialized.test_and_set()) {
        init_real_pthreads();
    }
    while (!function_wrappers_initialized);
    if (disableRain) {
        return real_pthread_mutex_lock(mutex);
    }
    int res = real_pthread_mutex_lock(&rain_lock);
    if (res) {
        // if unable to lock, just try not to disrupt host program
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: could not lock thread: %d\n", res);
        return real_pthread_mutex_lock(mutex);
    }
    bool found = false;
}
unsigned int t;
for (t = 0; !found && t < totalThreadCount; ++t) {
    if (clientThreads[t] && pthread_equal(*clientThreads[t], pthread_self())) {
        found = true;
    }
}

if (!found) {
    // commented out to prevent some programs from flooding this error
    // fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: requesting thread not
    // found, %p\n");
    res = real_pthread_mutex_unlock(&rain_lock);
    if (res) {
        // likely means mutex_lock does not own the mutex somehow
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: could not unlock
thread: %d\n", res);
        return real_pthread_mutex_lock(mutex);
    }
}

int ret = 0;
if (!pthread_mutex_trylock(mutex)) {
    // mutex acquired
    lockHolders[mutex] = t;
    ++mutexData[mutex].lockCount;
    mutexData[mutex].lockTimeStamp = timeNow();
} else {
    // mutex not acquired
    ++mutexData[mutex].contention;
    threadWaiters[t] = mutex;
    if (deadlockDetect(t, mutex)) {
        printf("DEADLOCK DETECTED\n");
        printf("mutex %p, thread %d, %p\n", mutex, t, clientThreads[t]);
        printBacktrace();
    }
    res = real_pthread_mutex_unlock(&rain_lock);
    if (res) {
        // likely means mutex_lock does not own the mutex somehow
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: could not unlock
thread: %d\n", res);
    }
    ret = real_pthread_mutex_lock(mutex);
    res = real_pthread_mutex_lock(&rain_lock);
    if (res) {
        // if unable to lock, just try not to disrupt host program
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: could not lock
thread: %d\n", res);
        return ret;
    }
    lockHolders[mutex] = t;
    ++mutexData[mutex].lockCount;
    mutexData[mutex].lockTimeStamp = timeNow();
    threadWaiters.erase(t);
}

27
res = real_pthread_mutex_unlock(&rain_lock);
if (res) {
    // likely means mutex_lock does not own the mutex somehow
    fprintf(stderr, "ERROR: RAIN: pthread_mutex_lock: could not unlock thread: %d\n", res);
    return ret;
}

int pthread_mutex_unlock(pthread_mutex_t *mutex) {
    if (!real_pthread_initialized.test_and_set()) {
        init_real_pthreads();
    }
    while (!function_wrappers_initialized);
    if (disableRain) {
        return real_pthread_mutex_unlock(mutex);
    }
    int res = real_pthread_mutex_lock(&rain_lock);
    if (res) {
        // if unable to lock, just try not to disrupt host program
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_unlock: could not lock thread: %d\n", res);
        return real_pthread_mutex_unlock(mutex);
    }
    int ret = real_pthread_mutex_unlock(mutex);
    lockHolders[mutex] = -1;
    mutexData[mutex].lockTimeTotal += timeNow() - mutexData[mutex].lockTimeStamp;
    real_pthread_mutex_unlock(&rain_lock);
    if (res) {
        // likely means mutex_lock does not own the mutex somehow
        fprintf(stderr, "ERROR: RAIN: pthread_mutex_unlock: could not unlock thread: %d\n", res);
    }
    return ret;
}
Appendix B: Relevant Man Pages

pthreads(7)
pthread_create(3)
pthread_join(3)
pthread_exit(3)
pthread_mutex(3)
pthread_equal(3)
pthread_sigqueue(3)
backtrace(3)
sigsetops(3)
sigaction(2)
getitimer(2)
dlopen(3)
clock_getres(2)
memset(3)
strerror(3)
printf(3)
References


