Development of a Deadlock-Detecting Resource Locking Algorithm for a Kernel Debugging User-space API Library (KDUAL)

TJHSST Computer Systems Lab
2004-2005

Timothy Wismer

June 20, 2005

Contents

List of Figures

Abstract

The kernel is the heart of an operating system; it is the program that is run when the computer first boots up, and it is responsible for accessing the computer hardware and performing other management tasks on behalf of all other programs. Because of its importance, the Linux Kernel must perform perfectly; any vulnerability, instability, or inefficiency will slow down or threaten the entire system.

Unfortunately, due to the nature of the program, kernel code cannot be easily debugged. The kernel runs in an environment called *kernel-space*, which is significantly different from the *user-space* environment in which ordinary programs run. The kernel provides user-space as an abstraction to the running programs, masking process scheduling, disk I/O, and so forth. Because the kernel expects to run in kernel-space, it cannot run in userspace. In addition, if there is an error while the kernel is running, it is difficult for the code to provide the tester with useful data about the error (because the kernel itself is responsible for access to the hard drives and monitor). The goal of the KDUAL project is to create a C library which implements the kernel Application Programming Interface (API) *in user-space* and performs automatic debugging. sections of kernel code can then be compiled against this library and run as ordinary programs for convenient testing.

This particular section of the project aims to implement the kernel's resource locking API, providing the core resource-locking algorithms with debugging code which will provide automatic detection, reporting, and resolution of deadlock situations, thus catching subtle locking errors in early testing and production and easing later debugging. Locking will be implemented in two parts–the core algorithms, with their own API designed to be most convenient for use in development, and simple wrapper code bridging that API to the kernel API. The core algorithms will also be suitable for applications other than kernel programming, e.g. as a generic lock-testing toolkit.

1 Background and Introduction

1.1 The Linux Kernel

The Linux Kernel performs a number of functions: device access, process management, memory management, and file systems, for example. Thus, it is a huge program (many thousands of lines of code). Moreover, because of the essential functionality it provides, the kernel must perform perfectly; any vulnerability, instability, or inefficiency can slow down or threaten the entire system—in the worst case scenario, bad kernel code can actually damage system hardware. Unfortunately, the very nature of kernel code also makes it difficult to debug. Kernels cannot be run as ordinary processes; testing a kernel requires compiling the new kernel and then reconfiguring and rebooting the test machine. In addition, if there is an error in the kernel code, it is difficult for the tester to obtain useful information about the error; first, because the kernel is responsible for all I/O operations, it is often impossible to interact with the system at all, and second, if the kernel code faults it will halt the whole system, leaving no way to analyze the crash. It is possible to run a kernel in a debugger, but only in a very limited fashion.

The KDUAL project intends to simplify kernel coding by creating a C library which implements the Kernel's Application Programming Interface (API) in user-space and provides an extensive debugging framework (for example, automatic deadlock checking in the locking implementation). Because this library will have an API identical to the real kernel, kernel programmers will be able to compile code against the KDUAL library for testing purposes without making any modifications. The resulting binary will run as an ordinary user-space program, so it can be run with minimal effort, will pose no threat to the system stability, and can be debugged using common tools such as the GNU Debugger (*gdb*). In addition, the library code will automatically provide the user with valuable debugging information.

Projects of this type have been produced before. The Arsenic [1] project transferred many of the traditionally OS-level operations of protocol management to the Network Interface Card (NIC), improving throughput and CPU efficiency. However, it is tied to a specific NIC and has dependencies on particular parts of the kernel code, both unnaceptable for a generic application-level implementation of kernel functionality. The Daytona [2] project is a user-space implementation of the TCP stack (which is responsible for analyzing packets and passing them to the appropriate application).

The Daytona TCP stack provides a valuable tool for studying and extending the TCP protocol, analyzing networks, or creating specialized user-level applciations. Alpine [3] is a similar but more expansive project that provides a network stack and virtual network driver in userspace (the driver stops short of actually communicating with the network device only because the Linux system requires such access to go through the kernel). Like KDUAL, both of these projects emphasize the need for transparency—providing a complete system which can be used as a substitute for the ordinary kernel code with few or no configuration changes to any of the affected applications or to the base system. Ideally, the code should be usable simply by re-linking applications to use the substitute library, with no changes to application code or the underlying kernel, and with no dependance on specific kernel code (so that the code will be portable across kernel versions). Unlike Arsenic, which focuses on optimizing performance, these projects also share the KDUAL goal of providing a system that is primarily a tool for debugging and development rather than a production system designed as an actual replacement for the kernel functionality (which is generally impossible when porting kernel code to user-space, because of the overhead incurred in user-space). However, because these projects operate as a functional, integral part of a working system (the user-level network stack is used for actual transmission of packets in a running system), they must also deal with issues of synchronization with the running kernel that do not occur in KDUAL (because the ersatz kernel provided by the KDUAL library is only as a test-harness for other programs and not to actually provide the kernel functionality).

1.2 Resource Locking

1.2.1 Basics of Locking

This particular part of the KDUAL project is focused on implementing the kernel resource-locking API. Resource-locking is essential for successful emphconcurrency: having multiple separate programs running simultaneously and working with the same resources—shared data objects, I/O handles (e.g. sockets), physical devices (reading data from a floppy disk), or any other resource which the processes cannot all use at once. In the absence of locking, such programs would simply "stomp" on each other, accessing the data simultaneously and potentially resulting in data corruption and program failure. Consider a simple example: a number i shared by two programs, A and B.

At some time during execution, i is 0, and each program wants to access it: A wants to set it to 3, and B wants to read its value for later use. If both processes access i without coordination, A will succeed in setting the value, but the value B returns is unpredictable. It may be 0 (if the read is completed before the write), 3 (if the write is completed before the read), or some other garbage value (if the read is performed while the write is occuring). To prevent an error situation, a process cannot safely operate on the data until it ensures that no other process is operating on the data. To achieve this, each resource is associated with a "lock" object. Generically, the lock has two states: locked and unlocked. To use a resource, a process must first obtain the associated lock. If the lock is currently in the unlocked state, the process can take the lock and then proceed to manipulate the data as it pleases, returning the lock to the unlocked state when it is done. If the lock is taken (because some other process is using the resource), the current process must wait on the lock until it is returned to the unlocked state (indicating that the resource is no longer in use). The process can then take the lock as before and proceed to use the resource.

The use of locks is illustrated in the classic "Dining Philosophers" thinking puzzle. Consider a group of philosophers seated around a table, with a plate and chopstick for each philosopher. A philosopher's life consists of two things: thinking and eating. Eating requires two chopsticks, so the philosopher must take *both* of the chopsticks next to him (one on either side). While thinking, a philosopher needs no chopstick. As long as the philosophers take turns thinking and eating, they should all be able to eat. However, a problem arises if all the philosophers attempt to eat at the same time. Each philosopher will grab one of the adjacent chopsticks, so that there are no chopsticks left on the table. Each philosopher will then wait for the philosopher next to him to give up his chopstick. Since no philosopher has two chopsticks, no philosopher will finish eating; therefore no philosopher will ever give up a chopstick, and no philosopher can ever have two chopsticks. Thus, the philosophers will starve to death. This can be avoided only by coordinating their actions; for example, access to the chopsticks could be controlled by a single lock, perhaps a bottle of hot sauce in the center of the table, with all the philosophers agreeing that they must take the lock before they can take chopsticks. When a philosopher is hungry and neither of his neighbors is eating, he takes the bottle, takes both chopsticks, puts the bottle back, and eats. When he is done he takes the bottle, yields the chopsticks, and puts the bottle back again. A philosopher will not attempt to get the bottle unless both chopsticks are available, and when he holds the bottle the other philosophers are prohibited by their agreement from interfering with his taking of the chopsticks. Since a philosopher cannot end up with only one chopstick, the starvation situation above has been avoided; in programming terminology, taking both chopsticks has become an *atomic* operation: it will either fail or succeed completely, and is guaranteed not to result in a partially altered state or to produce any intermediate states visible to other processes. When the overall set of actions that needs to be accomplished is not atomic (e.g. taking the chopsticks), resources can be safely accessed by binding that access to a single, atomic operation (e.g. taking the bottle).

1.2.2 Deadlock

Problems can arise in locking when one or more processes end up in *deadlock*: the processes “spin” forever while trying to take a lock, and are never able to take it, thus bringing the system to a halt. To illustrate this with the dining philosophers problem, suppose that there were *two* locks involved; perhaps one for the chopsticks and one for the food. To be able to eat, a philosopher must therefore hold both locks. The same problem occurs as with needing to hold both chopsticks: if two philosophers each get one lock, each will wait forever to get the other lock, again resulting in starvation. In computer programming, this condition is known as the *deadly embrace*. While the classic example involves only two processes and two locks, the same principle can be extended to any number of locks sought by any number of processes; deadlock occurs anytime a process cannot obtain the lock it is waiting for without giving up a lock it already holds. For example, the most trivial case of deadlock occurs when a process attempts to take a lock it already holds; each time the process checks the lock, it is in the locked state, so the process will keep waiting. However, it will never unlock the lock (since it is busy trying to lock it) and so will wait forever.

Deadlock causes all processes invovled to hang and makes the resource controlled by the lock unavaiable. Deadlock in the kernel is an especially severe problem, because it will hang the entire system—if the kernel locks up, userspace programs will also be unable to run, and the entire system becomes useless and requires reboot. In addition, deadlock can be a very difficult problem to identify and resolve, since it can involve multiple different processes and usually is not reliably reproducible (since it requires the processes to take their locks with very specific timing). This locking implementation will therefore include built-in testing for deadlock situations, vastly simplifying the debugging process and helping to produce more reliable code. It will also provide other valuable debugging features, e.g. status messages printed by certain segments of the locking code, which will help developers isolate and correct problems.

2 Theory

2.1 Kernel-space User-space Transition

A computer running the Linux Operating System can be viewed as a series of "layers", as in Fig. (1). Each layer is intended to be dependent only on the layers immediately adjacent to it. At the bottom is the machine hardware: processors, hard drives, video cards, RAM, and so on. Immediately above this is a layer of code called *drivers*. Each driver is an independent code module responsible for interacting with a single specific piece of hardware. Above the drivers is a layer of abstractions intended to mask the device-specific implementations of the drivers. For example, regardless of the type of device files are stored on (e.g. a local IDE or SCSI hard drive or a networked fileserver) and regardless of the type of filesystem present on the device (e.g. *ext2*,*ext3*,*ReiserFS*), processes will see the standardized system presented by the Virtual File System code. This allows a uniform method for operating on any files used by the system; the VFS dispatches instructions to the driver responsible for the particular device, which can then take whatever device-specific actions are appropriate. Finally, above this are the *syscalls* (for System Calls), which are the hooks intended to be invoked by regular user processes. These are the calls found in the second section of manpages as listed by `man syscalls`. As the manpage states, "The system call is the fundamental interface between an application and the Linux kernel." This exemplifies the nature of the kernel layers. To provide a simple example: a user program wants to write data to a file. It invokes the *write* syscall and knows that upon the call's completion the write has been accomplished. The *write* call tells the Virtual File System to write the data. The VFS in turn communicates with the appropriate filesystem driver; in the case of the TJCSL, the Andrews File System driver. This driver then communicates with the the driver for the actual physical device where the data will be written; this last driver is responsible for actually committing the data to

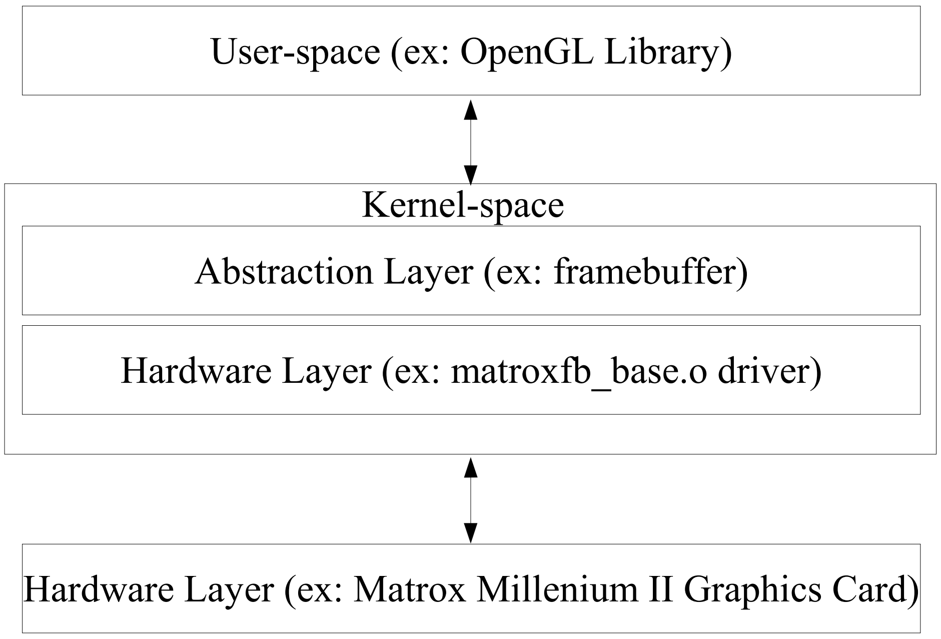
User-space (ex: OpenGL Library)
Kernel-space
Abstraction Layer (ex: framebuffer)
Hardware Layer (ex: matroxfb_base.o driver)
Hardware Layer (ex: Matrox Millenium II Graphics Card)

Figure 1: Operating Systems Layers

disk. However, this is all invisible to the original program. That program is completely unaware of any layer beneath the *write* call and the various other syscalls it uses to manipulate files; for all it knows, the system is pulling data from the ether.

Unfortunately, this structure makes debugging the kernel very difficult; since it is the bottom layer, all other layers ultimately depend on it, and any failure will have serious ramifications. Also, because the entire OS depends on the kernel, buggy code in the kernel can and in all likelihood will make the system unstable and/or insecure. This has a two-fold effect: first, it introduces the possiblity of data corruption or other system damage, and second, it prevents active debugging analysis after a crash. Moreover, it is difficult and somewhat unreliable to try to run the kernel in the *kdb* debugger. Typically, a kernel testing setup involves a seperate test machine with a serial console (another machine connected to it via serial cable that will allow access to the machine in the event of problems, e.g. a failure of the keyboard drivers). If there is a problem with the kernel, a few cryptic error messages will appear on the serial console and the system will become completely inoperable. This system of debugging is slow, requires additional hardware (a second computer and serial cable), and provides very little usable information.

The goal of KDUAL is to simplify this process by moving the code to be tested up from its ordinary layer into the top layer—user-space. Running in user-space, the code would not pose the same threat to system integrity and stability that it ordinarily would. The running kernel is unaffected by a code failure, so it remains stable, preserving the system and allowing post-crash debugging. In addition, code running as a normal process can be debugged using powerful, already-existing debuggers like *gdb*. This eliminates the need for additional hardware and provides vastly more infromation about the error. However, kernel code is dependant upon the adjacent layers to funtion: the *write* and *read* syscalls can hardly function without the filesystem provided by the VFS. As is the nature of the layered system, these are not available to user-space programs. Thus, the needed functionality must be re-implemented in user-space, preserving the existing kernel API so that switching from compilation against the user-space testing library to the actual kernel will be seamless. The KDUAL project will produce a user-space library of C code which implements the essential parts of the kernel API and which will provide the tester with built-in debugging checks and information that will further simplify the testing process.

2.2 Locking Implementation

2.2.1 Principles of Implementation

One of the key kernel components which must be implemented is resource locking: ensuring safe access to shared resources through the use of special objects (*locks*,*semaphores*,*mutexes*), as explained in the previous section. The focus of this project is completing the KDUAL implementation of the kernel's locking API, providing both the core locking algorithm and a powerful debugging infrastructure including automated deadlock detection.

Atomicity is the key feature of a locking system. If, in the example of the dining philosophers, the lock was *two* bottles of hot sauce, then two philosophers could each obtain one bottle, and end up fighting over the lock in the same manner they fought over the chopsticks. The value of a lock is that the operation of taking it is necessarily atomic; thus, when the lock is used to control more complicated actions (e.g. taking two chopsticks), the entire set of actions becomes atomic. If the lock is not taken, the entire sequence is immediately aborted, and the operation thus fails without changing the state. If the lock is taken, the other actions can be performed safely, because no one else can interfere with them until the lock is released; thus the change in state is not visible until the entire process completes. In the philosophers example, taking a single object is inherently atomic; we assume that the philosophers have the dignity to avoid fighting over the hot sauce. In programming, certain simple operations are guaranteed to be atomic because of the CPU architecture. The exact operations available differ among architectures, but all architectures provide *some* atomic operation or operations which make it possible to alter a value if and only if it matches another value; e.g. take a lock only if it is unlocked. Thus, the heart of any locking systems is a simple numerical value that can be atomically altered by individual CPU instructions.

Building a locking implementation from scratch would be tremendously difficult, and is unnecessary given the number of existing locking implementations. The kernel code itself provides a very simple locking implementation; this could theoretically be ported to user-space, but building on it to provide the additional functionality desired for the KDUAL locking implementation would be very difficult. The POSIX Threads (pthreads) library provides a complete, powerful user-space locking implementation, and is thus suitable for use in KDUAL. Naturally, however, it does not conform to the kernel API. The KDUAL locking implementation thus consists of a lock structure and a set of methods based around the pthreads library, providing more powerful debugging features with the Linux Kernel API. The KDUAL code will also help debugging by providing built-in checking for deadlock conditions where processes wait forever trying to obtain a lock. This checking will be accomplished by a dedicated deadlock-detection thread that monitors the dependencies of all the threads and responds to deadlock conditions.

2.2.2 Deadlock Detection

There are three basic strategies for dealing with the deadlock problem, as per Holliday [4]. In prevention, the protocol of the locking system inherently makes deadlock impossible (e.g. a system where a process is not allowed to hold resources while waiting on resources). This is a straightforward method of eliminating deadlock, but requires the system to be designed that way from the ground up. As such, it is not feasible for the KDUAL project because the project must replicate the API of the locking system used in the Linux Kernel, which is not a prevention system. The other two methods, avoidance and detection, *could* both be implemented in the KDUAL code because they would require modifications only to the internal implementation, not the API. However, an avoidance system, while preventing deadlock during testing, would provide no long-term benefit (when the code is integrated with the kernel proper) and as such could actually be concealing flaws that would then be revealed during production. A detection system will catch and deal with deadlocks in the running system and provide information which can be used by the developer to fix the code being tested so as to prevent the deadlock.

The detection thread relies on a structure called a Wait-For-Graph (WFG) as in Fig. (2), a common concept in deadlock detection. The *nodes* of the WFG represent processes in the system, and the (directed) *edges* of the graph are dependencies between processes, where an edge $i \rightarrow j$ indicates that process i is waiting for a resource currently held by j. If there is a deadlock, it will be detectable in the WFG as a *cycle*: a complete path from any node back to itself. The definition of deadlock is that the process is waiting on a resource that it cannot obtain without releasing a resource it already holds; this will necessarly result in a cycle in the WFG, because the process must (through some number of intermediaries) depend on itself. Thus there must be some series of edges leaving from a node and pointing back to the node. The reverse is also true: a cycle in the WFG always indicates a deadlock situation: the cycle indidcates that the process ultimately depends on itself, meaning that it is necessarily involved in deadlock and requires outside intervention. This means that checking for cycles in a WFG will detect all deadlocks *without* identifying non-deadlock situations as deadlocks (false positives); thus it is a reliable method for detection.

3 Design Criteria

A few others are beyond the scope of the project, in particular the scheduler, block devices, and the TCP/IP stack. The primary focus of the KDUAL project is on implementing the memory allocation algorithm, resource access controls such as spinlocks and semaphores, and the Virtual File System (VFS). Of these three, this project addresses resource control. As with all the components of the KDUAL system, the guiding principle is to implement

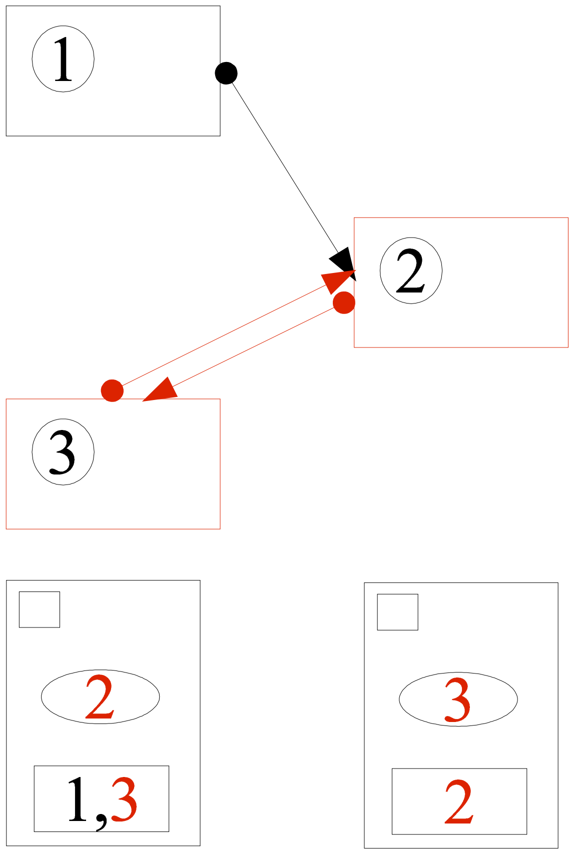

Figure 2: Wait-For-Graph
The Wait-For-Graph at top clearly indicates a deadlock via the cycle (in red) between processes 2 and 3. The lock situation from which the WFG is constructed is shown below: 2 and 3 wait on each other's locks, while process 1 is merely waiting on one of the locks and is not involved in the actual deadlock.

the functionality provided by the kernel.

All code will be written in C for speed and compatibility, and will follow the kernel style of taking advantage of special extensions for the GNU's Not UNIX (GNU) C compiler (*gcc*). Development will dependent on a few basic tools, such as the Vi IMproved (*VIM*) text editor and the GNU *make* utility for compiling code. The Concurrent Versions System (CVS) will be used to maintain the code repository, providing reliable archives and safe access for multiple developers. CVS in combination with the `cvsweb.cgi` perl script also serves as a convenient mechanism for making the code available online.

The ultimate test of a complete KDUAL system would primarily be ensuring that kernel code can successfully compile and run. Testing would require using code to test each function of the provided API, and running under various conditions (for example, low memory), to ensure proper performance. Likewise, testing of the locking system is primarily a test of the API, ensuring that the various functionality—e.g. creating and releasing locks—is complete.

4 Results and Future Development

The first stage of testing has been to confirm the viability of the basic methods (locking, unlocking, status checking, etc). So far, these tests have successfully completed, proving the capability of the locking system in a one-thread environment. The next tests will confirm the viability of the locking methods in a multiple-thread environment. The final testing phase will be to check the abilities of the deadlock-determination algorithm when it is complete.

The KDUAL library will greatly simplify and speed up the kernel development process. This will be an immediate benefit to the kernel development community. It will also have much more far-reaching effects, because a better kernel development produce will benefit all users of the kernel—a significant group including dedicated hackers, more casual users experimenting with non-Windows systems, schools, and even important businesses (including Microsoft's web hosts). In addition, the library and information about its development will provide a valuable basis for anyone attempting to undertake a similar project in kernel implementation—for example, producing a binary and library focused more on providing a viable production environment for "virtual servers" (like the User Mode Linux project).

References

[1] I. Pratt, K. Fraser, “Arsenic: A User-Accessible Gigagibt Ethernet Interface,” IEEE Infocom 2001, Available HTTP: http://www.cl.cam.ac.uk/Research/SRG/netos/arsenic/gige.ps

[2] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, E. Nahum, “Daytona - A User-Level TCP Stack,” Available HTTP: http://nms.lcs.mit.edu/~kandula/data/daytona.pdf

[3] D. Ely, S. Savage, D. Wetherall, “Alpine: A User-Level Infrastructure for Network Protocol Development,” Available HTTP: http://alpine.cs.washington.edu/alpineUsits01.pdf

[4] J. Holliday, A. El Abbadi, “Distributed Deadlock Detection,” Available HTTP: http://www.cse.scu.edu/~jholliday/dd_9_16.htm

[5] N. Krivokapić, A. Kemper, E. Gudes, “Deadlock Detection Agents: A Distributed Deadlock Detection Scheme,” Available HTTP: http://www.db.fmi.uni-passau.de/publications/techreports/MIP9617.ps.gz

APPENDIX

A lock.h

```
#ifndef KCORE_LOCK_H_
#define KCORE_LOCK_H_ 1

#include <kcore/stddef.h>
#include <kcore/types.h>

#include <pthread.h>

/*some includes that we need for the memory functions
 * these probably get included somewhere else in a full compile
 * but we need them for a standalone locking library
 */
#include <alloca.h>
#include <string.h>
#include <stdlib.h>

/*error code*/
#define KC_DEADLOCK 1
/*These ARE bitwise flags*/
#define KC_LOCK_DEFAULT 0x00        /*so we don't have to use 0 for flags */
#define KC_LOCK_RECURSIVE 0x01        /*allow recursive lock taking */
#define KC_LOCK_DEADLK_NOSEGV 0x02        /*return an error code instead of SEGVin
/*These are NOT BITWISE FLAGS*/
#define KC_STATUS_UNLOCKED 0x00
#define KC_STATUS_LOCKED 0x01
#define KC_STATUS_SEARCH 0x02

/*Debugging stuff: yes this should probably be system-wide not
 * just locking.  This is to provide information for debugging
 * of the library code itself, not applications that use it
 * Higher values for KC_LOCK_DEBUG activate more debugging
 * so higher values for "level"  mean lower importance.
 */
```

```
#ifndef KC_LOCK_DEBUG
#define KC_LOCK_DEBUG 0
#endif

/*macros for printing debug messages
 * dbg prints if debugging is on--same as dbg_pri(msg,1)
 * dbg_cond prints if cond is true
 * dbg_pri prints if the debugging level is high enough
 * dbg_pri_cond if the level is high enough and cond is true
 * eval_* runs arbitrary code as per the above restrictions
 * eval tacks a semicolon on to the end, so leave the last one off;
 */
#if KC_LOCK_DEBUG
#define dbg(msg)\
        do{\
                printf("**DEBUG**:");\
                printf(msg);\
                printf("\n");\
        }while(0)
#define dbg_cond(msg,cond)\
        do{\
                if((cond)){\
                        printf("**DEBUG**:");\
                        printf(msg);\
                        printf("\n");\
                }\
        }while(0)
#define dbg_pri(msg,level)\
        do{\
                if(KC_LOCK_DEBUG >= (level)){\
                        printf("**DEBUG**:");\
                        printf(msg);\
                        printf("\n");\
                }\
        }while(0)
#define dbg_pri_cond(msg,level,cond)\
        do{\
                if((KC_LOCK_DEBUG >= (level)) && (cond)){\
                        printf("**DEBUG**:");\
                        printf(msg);\
                        printf("\n");\
                }\
        }while(0)
#define dbg_eval(foo)\
        do{\
                foo;\
        }while(0)
#define dbg_eval_cond(foo,cond)\
        do{\
                if((cond)){\
                        foo;\
                }\
        }while(0)
#define dbg_eval_pri(foo,level)\
        do{\
                if(KC_LOCK_DEBUG >= (level)){\
                        foo;\
                }\
        }while(0)
#define dbg_eval_pri_cond(foo,level,cond)\
        do{\
                if((KC_LOCK_DEBUG >= (level)) && (cond)){\
                        foo;\
                }\
        }while(0)
#else
#define dbg(msg) do{}while(0)
#define dbg_cond(msg,cond) do{}while(0)
#define dbg_pri(msg,level) do{}while(0)
#define dbg_pri_cond(msg,level,cond) do{}while(0)
#define dbg_eval(msg) do{}while(0)
#define dbg_eval_cond(msg,cond) do{}while(0)
#define dbg_eval_pri(msg,level) do{}while(0)
#define dbg_eval_pri_cond(msg,level,cond) do{}while(0)
#endif
```

```
BEGIN_C_DECLS typedef struct kc_lock kc_lock_t;
struct kc_lock
{
  char magic1[4];
  pthread_mutex_t mutex;

  int status;

  pthread_t owner;
  kc_size_t count;

  pthread_cond_t lock;
  pthread_cond_t unlock;

  pthread_t search_thread;
  kc_lock_t *search_next;
  kc_lock_t *search_prev;
  kc_size_t search_refs;
  pthread_cond_t search_done;
  pthread_cond_t search_no_refs;
  char magic2[4];
};

/* Allows clients to see if the lock is in it's pure state
 * This is only for debugging of the locking library itself
 * Returns 1 if all variables in the lock match the values set by
 * kc_lock_create(), 0 otherwise
 */
#if KC_LOCK_DEBUG
int kc_lock_pure (kc_lock_t * lock);
#endif
/* Create a lock
 * returns errno on error
 * 0 on success
 */
int kc_lock_create (kc_lock_t ** lock);
```

```
/* Ways to obtain a lock */
int kc_lock_mustlock (int flags, kc_lock_t * firstlock, ...);
int kc_lock_lock (int flags, time_t timeout, kc_lock_t * firstlock,
                  ...);
int kc_lock_unlock (kc_lock_t * lock);

/* Status information */
int kc_lock_locked (kc_lock_t * lock);

/*Wait on lock condition*/
int kc_lock_waitlock (time_t timeout, kc_lock_t * lock);
int kc_lock_waitunlock (time_t timeout, kc_lock_t * lock);

/*Check for deadlock
 * Return EDEADLK on deadlock, 0 on success
 * timeout should be NULL to sleep forever
 * 0 to never sleep
 */
int kc_lock_check_deadlock (time_t * timeout, kc_lock_t * lock);
END_C_DECLS
#endif /* not KCORE_LOCK_H_ */
```

B lock.c

```
/*
 * Locking implementation based on the pthread_mutex_t providing automatic
 * deadlock checking
 *
 */

#include <stdarg.h>
#include <stdio.h>
#include <kcore/lock.h>
#include <pthread.h>
/*this should probably be OUR errno.h*/
#include <errno.h>
```

```
#define MAGIC1 "KcLk"
#define MAGIC2 "kClK"

#define recursive(x) (x & KC_LOCK_RECURSIVE)
#define nosegv(x) (x & KC_LOCK_DEADLK_NOSEGV)

/*Last lock in the list*/
static __thread kc_lock_t *last_lock = NULL;

/* Have the magic numbers changed? */
static inline void
check_magic (kc_lock_t * lock)
{
  SEGV_CHECK (memcmp (lock->magic1, MAGIC1, 4)
              || memcmp (lock->magic2, MAGIC2, 4));
}

/* create a lock
 * returns errno on error
 * 0 on success */
int
kc_lock_create (kc_lock_t ** lock)
{
  pthread_mutexattr_t err_attr;
  pthread_mutexattr_init (&err_attr);
  pthread_mutexattr_settype (&err_attr, PTHREAD_MUTEX_ERRORCHECK_NP);
  pthread_mutex_t templ_mutex;
  pthread_mutex_init (&templ_mutex, &err_attr);
  //pthread_mutex_t templ_mutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
  pthread_cond_t templ_cond = PTHREAD_COND_INITIALIZER;
  *lock = (kc_lock_t *) malloc (sizeof (**lock));
  if (!*lock)
    return errno;
  memcpy ((*lock)->magic1, MAGIC1, 4);
  (*lock)->status = KC_STATUS_UNLOCKED;
  (*lock)->count = 0;
  (*lock)->search_prev = NULL;
  (*lock)->search_next = NULL;
  (*lock)->mutex = templ_mutex;
  (*lock)->lock = templ_cond;
  (*lock)->unlock = templ_cond;
  (*lock)->search_done = templ_cond;
  (*lock)->search_no_refs = templ_cond;
  (*lock)->search_refs = 0;
  memcpy ((*lock)->magic2, MAGIC2, 4);
  return 0;
}

#if KC_LOCK_DEBUG
int
kc_lock_pure (kc_lock_t * lock)
{
  check_magic (lock);
  if (lock->status != KC_STATUS_UNLOCKED)
    return 0;
  if (lock->count != 0)
    return 0;
  if (lock->search_prev != NULL)
    return 0;
  if (lock->search_next != NULL)
    return 0;
  if (lock->search_refs != 0)
    return 0;
  return 1;
}
#endif

/* Lock with infinite timeout */
int
kc_lock_mustlock (int flags, kc_lock_t * firstlock, ...)
{
  int status;
  va_list ap;
  kc_lock_t *lock;
  kc_size_t count = 1;
  kc_lock_t **locklist;
  dbg_pri ("Reached kc_lock_mustlock", 2);
#if 0
  va_start (ap, firstlock);
  while (lock = va_arg (ap, kc_lock_t *))
    count++;
  va_end (ap);
  locklist = alloca (count * sizeof (*locklist));
  va_start (ap, firstlock);
  /* sort the locks in memory order so they can be safely taken */
  while (lock = va_arg (ap, kc_lock_t *));
  va_end (ap);
#endif
  /*begin the process of actually taking the locks */
  /* foreach lock */
  va_start (ap, firstlock);
  lock = firstlock;
  do
    {
      kc_lock_t *my_last_lock = last_lock;
      dbg_pri ("Entered locktaking while loop (in kc_lock_mustlock)",
               2);
      /*SEGV_ON(status = pthread_mutex_lock(&my_last_lock->mutex));
         while(my_last_lock->search_refs !=0){
         SEGV_ON(status=pthread_cond_wait(&my_last_lock->search_no_refs,&my_last_l
         } */
      SEGV_ON (status = pthread_mutex_lock (&lock->mutex));
      /* WE NOW HOLD THE MUTEX */
      /* easy case: take the lock */
      if (lock->status == KC_STATUS_UNLOCKED)
        {
          lock->owner = pthread_self ();
          lock->status = KC_STATUS_LOCKED;
          lock->count++;
          /* something's wrong */
          SEGV_ON (lock->search_refs != 0);
          SEGV_ON (pthread_mutex_unlock (&lock->mutex));
          continue;
        }
      dbg ("The lock was not unlocked; getting ready to wait");
      /* recursive case */
      if (lock->owner == pthread_self ())
        {
          dbg_cond
            ("You have attempted to take a lock twice without the recursion flag!'
             !recursive (flags));
          SEGV_ON (!recursive (flags));
          lock->count++;
          SEGV_ON (pthread_mutex_unlock (&lock->mutex));
          continue;
        }
      /* We actually have to wait on the lock */
      /* We have to loop in case we don’t snag the lock after waiting:
       * need to redo deadlock check before doing anything else
       * again, WE HOLD THE MUTEX BUT NOT THE LOCK */
      while (lock->owner != pthread_self ())
        {
          /*FIXME: all of this should get altered so that
           * the codnitions for the deadlock checking routine are
           * uniform
           * TODO:
           * check for legitimate conditions
           * call deadlock
           * decide based on return value
           * last_lock->next MUST ALREADY BE SET */
          pthread_t prev_checker = lock->search_thread;
          /* We get to search */
          if ((lock->status != KC_STATUS_SEARCH) ||
              (lock->search_thread > pthread_self ()))
            {
              lock->search_thread = pthread_self ();
              /*deadlock check! */
              lock->search_thread = prev_checker;
              pthread_cond_broadcast (&lock->search_done);
              /* we can wait for the lock now */
              pthread_cond_wait (&lock->unlock, &lock->mutex);
```

```
              if (lock->status == KC_STATUS_UNLOCKED)
                {
                  lock->owner = pthread_self ();
                  lock->status = KC_STATUS_LOCKED;
                  lock->count++;
                  SEGV_ON (pthread_mutex_unlock (&lock->mutex));
                }
            }
          else
            {
              /*wait for lower-ranked checkers to be done */
              while ((lock->status == KC_STATUS_LOCKED) &&
                     (lock->search_thread < pthread_self ()))
                pthread_cond_wait (&lock->search_done, &lock->mutex);
              lock->search_thread = pthread_self ();
              /* deadlock check! */
              lock->search_thread = prev_checker;
              pthread_cond_broadcast (&lock->search_done);
              /* we can wait for the lock now */
              pthread_cond_wait (&lock->unlock, &lock->mutex);
              if (lock->status == KC_STATUS_UNLOCKED)
                {
                  lock->owner = pthread_self ();
                  lock->status = KC_STATUS_LOCKED;
                  lock->count++;
                  SEGV_ON (pthread_mutex_unlock (&lock->mutex));

                }
            }
        }
    }
  while (lock = va_arg (ap, kc_lock_t *));
  va_end (ap);
  return 0;
}

/* Lock with timeout*/
int
```

```
kc_lock_lock (int flags, time_t timeout, kc_lock_t * firstlock, ...)
{
  /*FIXME: this code is just a copy of kc_lock_mustlock with
   * the commented junk and deadlock-checking stuff removed and
   * timeouts added.  It is a temporary solution because I needed
   * the timeout functionality; once kc_lock_mustlock() is put in
   * decent shape, this should change to reflect that.
   * (Also it will let pthread_mutex_lock() run forever)
   */
  int status;
  va_list ap;
  kc_lock_t *lock;
  kc_size_t count = 1;
  kc_lock_t **locklist;
  struct timespec stoptime;
  stoptime.tv_sec = time (NULL) + timeout;
  stoptime.tv_nsec = 0;
  dbg_pri ("Reached kc_lock_lock", 2);
  /* foreach lock */
  va_start (ap, firstlock);
  lock = firstlock;
  do
    {
      kc_lock_t *my_last_lock = last_lock;
      dbg_pri ("Entered locktaking while loop (in kc_lock_lock)", 2);
      SEGV_ON (status = pthread_mutex_lock (&lock->mutex));
      /* WE NOW HOLD THE MUTEX */
      /* easy case: take the lock */
      if (lock->status == KC_STATUS_UNLOCKED)
        {
          lock->owner = pthread_self ();
          lock->status = KC_STATUS_LOCKED;
          lock->count++;
          /* something's wrong */
          SEGV_ON (lock->search_refs != 0);
          SEGV_ON (pthread_mutex_unlock (&lock->mutex));
          continue;
        }
      dbg ("The lock was not unlocked; getting ready to wait");
      /* recursive case */
      if (lock->owner == pthread_self ())
        {
          dbg_cond
            ("You have attempted to take a lock twice without the recursion flag!'
             !recursive (flags));
          SEGV_ON (!recursive (flags));
          lock->count++;
          SEGV_ON (pthread_mutex_unlock (&lock->mutex));
          continue;
        }
      /* We actually have to wait on the lock */
      /* We have to loop in case we don't snag the lock after waiting:
       * need to redo deadlock check before doing anything else
       * again, WE HOLD THE MUTEX BUT NOT THE LOCK */
      while (lock->owner != pthread_self ())
        {
          pthread_cond_timedwait (&lock->unlock, &lock->mutex,
                                  &stoptime);
          if (lock->status == KC_STATUS_UNLOCKED)
            {
              lock->owner = pthread_self ();
              lock->status = KC_STATUS_LOCKED;
              lock->count++;
              SEGV_ON (pthread_mutex_unlock (&lock->mutex));
            }
        }
    }
  while (lock = va_arg (ap, kc_lock_t *));
  va_end (ap);
  return 0;
}

/*Release Lock
 * The worst case scenario here is that someone is in the process of
 * deadlock-checking this lock.
 * */
```

```
int
kc_lock_unlock (kc_lock_t * lock)
{
  dbg ("Reached kc_lock_unlock");
  SEGV_ON (pthread_mutex_lock (&lock->mutex));
  if (lock->owner != pthread_self ())
    return -1;
  dbg ("Checked to make sure we own the lock");
  /*check lock->count to handle recursive takes
   * don't actually unlock it until there are no
   * recursive takes*/
  dbg_eval (printf
            ("Lock count is %d in call to kc_lock_unlock()\n",
             lock->count));
  if (lock->count)
    lock->count--;
  if (lock->count)
    {
      SEGV_ON (pthread_mutex_unlock (&lock->mutex));
      return 0;
    }
  SEGV_ON (pthread_mutex_unlock (&lock->mutex));
  lock->status = KC_STATUS_UNLOCKED;
  /*do something w/ lock->search_refs here? */
  lock->search_next = lock->search_prev = NULL;
  //pthread_cond_broadcast(&lock->unlock);
  dbg ("Unlocked the lock");
  return 0;
}

/* Check lock status*/
int
kc_lock_locked (kc_lock_t * lock)
{
  SEGV_ON (pthread_mutex_lock (&lock->mutex));
  if (lock->status == KC_STATUS_UNLOCKED)
    {
      SEGV_ON (pthread_mutex_unlock (&lock->mutex));
      return 0;
    }
  SEGV_ON (pthread_mutex_unlock (&lock->mutex));
  return 1;
}

/* Wait for condition without affecting the lock*/
int kc_lock_waitlock (time_t timeout, kc_lock_t * lock);
int kc_lock_waitunlock (time_t timeout, kc_lock_t * lock);

/* Check for a deadlock condition
 * If timeout is 0 never wait on a lock -- if it is NULL wait forever
 * Returns 0 on success, KC_DEADLK if there is deadlock */
int
kc_lock_check_deadlock (time_t * timeout, kc_lock_t * start)
{
#if 0
  /*TODO:
   * Precond: we hold the mutex on start
   * we are allowed to be searching (i.e. there is no lower-ranked searcher)
   * last_lock->next points to the next lock in the chain
   * Steps:
   * check for ownership--if it belongs to us, it's deadlock
   * check the next pointer--if it's null, it's all good
   * set search_thread and status (preserving old values)
   * up the refcount <- this doesn't matter anymore
   * yield the mutex and wait on the next mutex
   * wait until we are allowed to search on that mutex
   * RECURSE
   * spin on start's mutex until we are the current search thread
   * if we were not the current search thread, somebody supplanted us,
   * so we should go back up-> i.e. recurse again.  and again.  and again.
   * WE SHOULD NOT GO BACK UP IF THE RECURSIVE CALL RETURNED AN ERROR
   * restore the old status and thread
   * return (the status of the recursive call)
   * On deadlock:
   * this is messy because we don't want to fall back along the list until
   * the chain is broken.  So we keep track of the current mutex, grab last_lock,
   * make the pointer null, then return to the end of the chain and start falling
   * REFCOUNT nonsense with breaking the pointer.
   */
  if (start->owner == pthread_self ())
    {
      /*we're screwed, do the deadlock thing */
      return KC_DEADLK;
    }
  if (!start->search_next)
    return 0;                        /*end of list! */
  /*we have to go to the next lock
   * so preserve the data*/
  int status = start->status;
  pthread_t search_thread = start->search_thread;
  start->status = KC_STATUS_SEARCH;
  start->search_thread = pthread_self ();
  start->search_refs++;
  /*yield the lock */
  SEGV_ON (pthread_mutex_unlock (&start->mutex));
  kc_lock_t *next = start->search_next;
  SEGV_ON (pthread_mutex_lock (&next->mutex));
  int loop = 0;
  do
    {
      /*wait for searchers if needed */
      while ((next->status == KC_STATUS_SEARCH)
             && (next->search_thread < pthread_self ()))
        pthread_cond_timedwait (&next->search_done, &next->mutex,
                                /*FIXME*/);
      /*recurse! */
      int failed = kc_lock_check_deadlock (timeout, next);
      if (failed)
        {
          /*dothatcrazydeadlockthang */
          return KC_DEADLK;
        }
      /*wait for searchers who passed us to finish */
      while (next->search_thread != pthread_self ())
```

```
        pthread_cond_wait (&next->search_done, &next->mutex);
    }
  while (!loop);
#endif
  return 0;
}
```

C lock_test.c

```
#include <stdlib.h>
#include <stdio.h>
#include <kcore/lock.h>

void simple_tests ();

int
main (void)
{
  pthread_t thread1, thread2, thread3;
  pthread_create (&thread1, NULL, (void *) simple_tests, NULL);
  pthread_create (&thread2, NULL, (void *) simple_tests, NULL);
  pthread_create (&thread3, NULL, (void *) simple_tests, NULL);
  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);
  pthread_join (thread3, NULL);
  return 0;
}

void
simple_tests ()
{
  kc_lock_t *lock1, *lock2;        /*locks yay */
  int err = 0;
  if ((err = kc_lock_create (&lock1)))
    {
      printf ("%d: Error %d creating lock\n", pthread_self (), err);
      exit (1);
    }
  if ((err = kc_lock_create (&lock2)))
    {
      printf ("%d: Error %d creating lock\n", pthread_self (), err);
      exit (1);
    }

  /*lock seperately with kc_lock_mustlock */
  printf ("%d: Attempting to take locks one and two seperately\n",
          pthread_self ());
  kc_lock_mustlock (KC_LOCK_DEFAULT, lock1, NULL);
  kc_lock_mustlock (KC_LOCK_DEFAULT, lock2, NULL);
  printf ("%d: Calls succeeded, checking status of both locks\n",
          pthread_self ());
  if (!kc_lock_locked (lock1))
    {
      printf ("%d: ERROR: lock1 appeared to be unlocked\n",
              pthread_self ());
      exit (1);
    }
  if (!kc_lock_locked (lock2))
    {
      printf ("%d: ERROR: lock2 appeared to be unlocked\n",
              pthread_self ());
      exit (1);
    }
  printf ("%d: Both locks appear locked\n", pthread_self ());
  printf ("%d: Attempting to release locks one and two\n",
          pthread_self ());
  if (kc_lock_unlock (lock1))
    {
      printf ("%d: ERROR: failed to unlock lock1\n", pthread_self ());
      exit (1);
    }
  if (kc_lock_unlock (lock2))
    {
      printf ("%d: ERROR: failed to unlock lock2\n", pthread_self ());
      exit (1);
    }
  printf ("%d: Calls succeeded, checking status of both locks\n",
          pthread_self ());
  if (kc_lock_locked (lock1))
    {
      printf ("%d: ERROR: lock1 appeared to be locked\n",
              pthread_self ());
      exit (1);
    }
  if (kc_lock_locked (lock2))
    {
      printf ("%d: ERROR: lock2 appeared to be locked\n",
              pthread_self ());
      exit (1);
    }
  printf ("%d: Both locks appear unlocked\n", pthread_self ());

  /*lock together with kc_lock_mustlock */
  printf ("%d: Attempting to take locks one and two in one call\n",
          pthread_self ());
  kc_lock_mustlock (KC_LOCK_DEFAULT, lock1, lock2, NULL);
  printf ("%d: Call succeeded.  Checking lock status\n",
          pthread_self ());
  if (!kc_lock_locked (lock1))
    {
      printf ("%d: ERROR: lock1 appeared to be unlocked\n",
              pthread_self ());
      exit (1);
    }
  if (!kc_lock_locked (lock2))
    {
      printf ("%d: ERROR: lock2 appeared to be unlocked\n",
              pthread_self ());
      exit (1);
    }
  printf ("%d: Both locks appear locked\n", pthread_self ());
  printf ("%d: Attempting to release locks one and two\n",
          pthread_self ());
```

```
if (kc_lock_unlock (lock1))
  {
    printf ("%d: ERROR: failed to unlock lock1\n", pthread_self ());
    exit (1);
  }
if (kc_lock_unlock (lock2))
  {
    printf ("%d: ERROR: failed to unlock lock2\n", pthread_self ());
    exit (1);
  }
printf ("%d: Calls suceeded.  Checking lock status\n",
        pthread_self ());
if (kc_lock_locked (lock1))
  {
    printf ("%d: ERROR: lock1 appeared to be locked\n",
            pthread_self ());
    exit (1);
  }
if (kc_lock_locked (lock2))
  {
    printf ("%d: ERROR: lock2 appeared to be locked\n",
            pthread_self ());
    exit (1);
  }
printf ("%d: Both locks appear unlocked\n", pthread_self ());
/*recursive lock taking? */
printf ("%d: Attempting to take lock1 twice (recursively)\n",
        pthread_self ());
if (kc_lock_mustlock (KC_LOCK_DEFAULT, lock1, NULL))
  {
    printf ("%d: ERROR: failed to lock lock1 (standard locking)\n",
            pthread_self ());
    exit (1);
  }
if (kc_lock_mustlock (KC_LOCK_RECURSIVE, lock1, NULL))
  {
    printf ("%d: ERROR: failed to lock lock1 (recursively)\n",
            pthread_self ());
```

```
      exit (1);
    }
  if (kc_lock_unlock (lock1))
    {
      printf ("%d: ERROR: failed to unlock lock1\n", pthread_self ());
      exit (1);
    }
  /*lock1 should still be locked */
  if (!kc_lock_locked (lock1))
    {
      printf
        ("%d: ERROR:lock 1 appeared unlocked after the first release\n",
         pthread_self ());
      exit (1);
    }
  if (kc_lock_unlock (lock1))
    {
      printf ("%d: ERROR: failed to unlock lock1\n", pthread_self ());
      exit (1);
    }
  /*try to take lock twice without recursion */
/*        printf("%d: Attempting to take lock1 twice (without recursion)--THIS SHO
        if(kc_lock_mustlock(KC_LOCK_DEFAULT,lock1,NULL)){
                printf("%d: ERROR: failed to lock lock1 (standard locking)\n",pthr
                exit(1);
        }
        if(kc_lock_mustlock(KC_LOCK_DEFAULT,lock1,NULL)){
                printf("%d: ERROR: failed to lock lock1 (standard locking)\n",pthr
                exit(1);
        }
        printf("%d: ERROR: taking lock1 twice did not segfault\n",pthread_self());
  /*print success and exit */
  printf ("Completed test\n");
}
```

D lock_test_output

Results from the tests in *lock_test.c*. The number preceding each message is the ID of the thread printing the statement.

```
16386: Attempting to take locks one and two seperately 16386: Calls succeeded, che
  unlocked;
     getting ready to
       wait **
       DEBUG **:
       Reached
       kc_lock_unlock **
       DEBUG **:
       Checked
       to
       make
       sure
       we
       own
       the
       lock
       Lock
       count
       is
       2
       in
       call
       to
     kc_lock_unlock () **
  DEBUG **:
  Reached
  kc_lock_unlock **
  DEBUG **:
  Checked
  to
  make
  sure
  we
  own
  the
  lock
  Lock
  count
  is
  1
  in
  call
  to
kc_lock_unlock () **
DEBUG **:Unlocked the lock
  Completed test
  49156:Attempting to take locks one and two seperately
  49156:Calls succeeded, checking status of both locks
  49156:Both locks appear locked
  49156:Attempting to release locks one and two
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to
kc_lock_unlock () **
DEBUG **:Unlocked the lock
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to
kc_lock_unlock () **
DEBUG **:Unlocked the lock
  49156:Calls succeeded, checking status of both locks
  49156:Both locks appear unlocked
  49156:Attempting to take locks one and two in one call
  49156:Call succeeded.Checking lock status
  49156:Both locks appear locked
  49156:Attempting to release locks one and two
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to
kc_lock_unlock () **
DEBUG **:Unlocked the lock
```

```
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to
kc_lock_unlock () **
DEBUG **:Unlocked the lock
  49156:Calls suceeded.Checking lock status
  49156:Both locks appear unlocked 49156:Attempting to take lock1
twice (recursively) **
DEBUG **:The lock was not unlocked;
     getting ready to wait
       ** DEBUG **:Reached kc_lock_unlock
       ** DEBUG **:Checked to make sure we own the lock
       Lock count is 2 in call to kc_lock_unlock ()
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to kc_lock_unlock ()
  ** DEBUG **:Unlocked the lock
  Completed test
  32771:Attempting to take locks one and two seperately
  32771:Calls succeeded, checking status of both locks
  32771:Both locks appear locked
  32771:Attempting to release locks one and two
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to kc_lock_unlock ()
  ** DEBUG **:Unlocked the lock
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
  Lock count is 1 in call to kc_lock_unlock ()
  ** DEBUG **:Unlocked the lock
  32771:Calls succeeded, checking status of both locks
  32771:Both locks appear unlocked
  32771:Attempting to take locks one and two in one call
  32771:Call succeeded.Checking lock status
  32771:Both locks appear locked
  32771:Attempting to release locks one and two
  ** DEBUG **:Reached kc_lock_unlock
  ** DEBUG **:Checked to make sure we own the lock
```

```
Lock count is 1 in call to kc_lock_unlock ()
** DEBUG **:Unlocked the lock
** DEBUG **:Reached kc_lock_unlock
** DEBUG **:Checked to make sure we own the lock
Lock count is 1 in call to kc_lock_unlock ()
** DEBUG **:Unlocked the lock
32771:Calls suceeded.Checking lock status
32771:Both locks appear unlocked
32771:Attempting to take lock1 twice (recursively)
** DEBUG **:The lock was not unlocked;
   getting ready to wait
     ** DEBUG **:Reached kc_lock_unlock
     ** DEBUG **:Checked to make sure we own the lock
     Lock count is 2 in call to kc_lock_unlock ()
** DEBUG **:Reached kc_lock_unlock
** DEBUG **:Checked to make sure we own the lock
Lock count is 1 in call to kc_lock_unlock ()
** DEBUG **:Unlocked the lock Completed test
```