Summary of:

Threads Cannot Be Implemented As a Library

Paper by Hans-J Boehm
Summary by Håkan Lundvall

A lot of multi-threaded code is developed in languages that were designed without thread support. In order to handle concurrency issues the programmers are directed to the use of libraries such as Pthreads. In this paper the author points out some issues that he claims makes assuring a multi-threaded program’s correctness not possible.
In order to reason about the correctness of a multi-treaded program there must be a specified memory model that describes what happens when code referring to the same variables runs concurrently. Languages like C and C++ lack a specified memory model.
Pthreads attack this problem by stating that all write accesses to shared variables must be done between calls to the library primitives; *pthread_mutex_lock()* and *pthread_mutex_unlock()* or similar functions. The idea being that since the compiler knows nothing of the functions and must treat them as opaque functions, thus the compiler must make sure all variables must be written back to memory before the function call is made and not move memory operations around such function calls.
Three potential issues regarding program correctness is presented in the paper; *concurrent modification*, *rewriting of adjacent* data and *register promotion*.

Concurrent modification

The Pthreads specification prohibits access to a shared variable while another thread is modifying it, but whether this can be assured or not depends on the semantics of the underlying language of which the library specification ha no control. Since a language designed without concurrency in mind might transform code in such a way that the semantics of the program remain the same in a single threaded context. As an example, two threads each executing one of the following statements where x and y are initially zero:

```
if (x == 1) ++y;
if (y == 1) ++x;
```

could be transformed to:

```
++y; if (x != 1) --y;
++x; if (y != 1) --x;
```

In a single threaded environment this would not change the semantics, but when there are two threads involved there is a race since the shared variables are altered without using the locking primitives of the Pthreads library.

Rewriting of Adjacent Data

The second issue brought up regards the fact that the compiler might write to adjacent memory locations while storing a variable to memory. In fact, in some situations it might be unavoidable; consider a struct containing bit fields of the form:

```
struct { int a:17;  int b:15 } x;
```

An assignment to the variable `x.a` is likely to be implemented as a 32 bit wide store instruction causing the value of `x.b` to be rewritten, which is alright for sequential code, but if there is a concurrent update to `x.b` between the old value of `x` is read and the new value of `x.a` is written, the update of `x.b` is lost.
For bit fields this is a well recognized fact and does not actually violate the Pthreads standard since only concurrent writes to "memory locations" is prohibited by it. But, in general there is nothing that prohibits a compiler to rewrite adjacent data in other situations as well.

Register promotion

Optimizing compilers often promote frequently accessed variables to registers. Consider the following program:

```
for (...) {
  ...
  if (mt) pthread_mutex_lock(...);
  x = ... x ...
  if (mt) pthread_mutex_unlock(...);
}
```

For performance reasons the programmer wants to avoid the extra overhead of calling the lock functions if there is only one active thread. Since `x` is accessed inside a loop and the compiler might speculate that `mt` is most often false, this could be transformed to the following:

```
r = x;
for (...) {
  ...
  if (mt) {
    x = r; pthread_mutex_lock(...); r = x;
  }
  r = ... r ...
  if (mt) {
    x = r; pthread_mutex_unlock(...); r = x;
  }
}
x = r;
```

Extra reads and writes to a shared variable have been introduced outside the lock and the code is no longer safe.

Performance

The only style of programming allowed by the Pthreads library is one where all modifications of shared variables are protected by mutex locks, since the calls to these library routines are supposed to add memory barriers so that reordering of memory operations outside these barriers are harmless.
These library calls introduce a significant overhead due to expensive atomic operation instructions and memory barriers as well as the normal overhead of a library call.
There is a wide range of lock-free and wait-free programming techniques, avoiding much of this overhead, that are no allowed should the programming rules of Pthreads be followed. The

paper contains two examples of algorithms that are slowed down considerably by the use of locks; a parallel Sieve of Eratosthenes and a parallel mark-sweep garbage collector. Both showing that a mutex locked implementation using four threads can be even slower than a sequential run where no locks are needed.