

Threads Cannot Be Implemented As a Library

Authored by Hans J. Boehm

Presented by Sarah Sharp

February 18, 2008

Outline

POSIX Thread Library Operation

Vocab

Problems with pthreads

POSIX Thread Library

- Many languages were not originally specified with thread support.
- For C and C++, the POSIX Thread Library (pthreads) was created for thread support.
- The authors argue that issues with thread support "lie almost exclusively with the compiler and the language specification itself, not with the thread library or its specification. Hence they cannot be addressed purely within the thread library."

What does the POSIX Thread Library guarantee?

- "Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers."
- No thread can read or modify a memory location while another thread may be modifying it.
- Functions to "synchronize memory with respect to other threads".
 - `pthread_mutex_lock()`
 - `pthread_mutex_unlock()`
- These calls include memory barriers to prevent hardware reordering of memory operations around these calls
- The compiler must assume these functions can read or write any global variable, so memory references can't be moved across the call to this function.

Vocab

- Sequential
- Sequential Consistency
- Sequential Correctness
 - if the program were executed in a sequential manner (i.e. one thread on one processor) with no other programs being executed, it would produce sequentially correct results.
- compiler or hardware transformations of multiprocess algorithms may preserve "sequential correctness" of individual processes but not preserve "sequential consistency" of the whole algorithm.

Compiler Optimizations

Program 1 (x and y initialized to 0):

Thread 1: if (x == 1) ++y;

Thread 2: if (y == 1) ++x;

Can we ever get x == 1 and y == 1?

Compiler Optimizations

Program 1 (x and y initialized to 0):

Thread 1: if (x == 1) ++y;

Thread 2: if (y == 1) ++x;

Can we ever get x == 1 and y == 1?

What if the compiler does this?

Thread 1: ++y; if (x != 1) -y;

Thread 2: ++x; if (y != 1) -x;

This is a "sequentially correct" result, but a multithreaded system running this code will not be "sequentially consistent".

Solution: The compiler must know that this program will run in a multithreaded environment. The library itself does not ensure the optimizer produces correct code.

Memory Operations on Adjacent Addresses

- Processors cannot write or read memory of arbitrary widths.
- A write of `x.a = 42` in a structure like this

```
struct {  
    int a:17; // a is a bitfield, 17 bits wide  
    int b:15; // b is a bitfield, 15 bits wide  
} x;
```

- may involve reading and writing more than just the variable `a`:

```
int temp = x;  
temp &= ~0x1ffff; // mask off old a  
temp |= 42;  
x = temp; // overwrite all of x
```

- What if we have one thread that updates `a` in structure `x`, and another thread that updates `b` in structure `x`?

Memory Operations on Adjacent Addresses

- The POSIX threads specification prohibits concurrent writes to the same memory location, so it endorses this behavior.
- May cause problems if different fields were supposed to be protected by different locks.
- The pthreads specification also allows this behavior for global variables adjacent to a structure.
- The linker may reorder global variables in memory, so no variable is safe!
- Solution: Make the language specify when adjacent data can be implicitly overwritten.

Register promotion

Several compiler optimizations lead to this problem:

- Instead of reading a variable repeatedly inside a loop, store it in a register.
- The conditional branch on an `if` is rarely taken, so optimize for the case where it is not taken.

Register promotion

The code in question (where x is a global variable):

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

Register promotion

The optimized code:

```
r = x;
for (...) {
  ... // might change r in here
  if (mt) {
    x = r; // update x with current value of the register
    pthread_mutex_lock(...);
    r = x; // update r in case something changed x
  }
  r = ... r ...
  if (mt) {
    x = r;
    pthread_mutex_unlock(...);
    r = x;
  }
}
```

Register promotion

- Why do this?
- But the pthread standard says that memory must be "synchronized with" the logical program state around the `pthread_mutex_lock()` and `pthread_mutex_unlock()` calls.
- If `x` is protected by a lock, we are reading and writing it outside the lock.
- Solution: Make sure the compiler is aware this could be a threaded program. It won't do register promotion around unknown function calls.

What about lockless algorithms?

- The only parallel programming technique the POSIX thread standard allows is mutual exclusion.
- This excludes lockless algorithms and algorithms that use atomic operations (but not locks).
- This is part of the reason pthreads (almost) work.
- However, this is very costly...

pthread vs. lockless algorithms