Threads Cannot Be Implemented As a Library

or how C Pthreads / user library threading is broken

Hans-J. Boehm

Presented for CS533 @ PSU by Larry Diehl
Thesis

• Threads cannot be implemented as a user library

• Threads must be part of the programming language specification
Outline

• Course context
• Motivation
• Specifications
• 3 Specification Bugs
• Performance
• Conclusion
Course Context

• Heavyweight kernel threads
• Lightweight user threads
• Swap out scheduling / threading library
• Small static number of threads
• Large dynamic number of threads
• Problems for C language
• More generally, could use different langs to preserve the swapping “dream”
Motivation for Threads

• Multithreaded programs pervasive
• Word processor character input & spellchecking
• Background processing & multiple GUI windows rendering
• Multiprocessors now common
• Utilize more processing power
Programming Language Specification

- Written in formal logic using judgements
- Model theoretic or syntactic abstraction of local context, definitions, state, types, threads, etc
- Semantics define how these interact
- Defines a memory model
- Concurrent execution is part of the formal semantics; e.g. Java
Unspecified Parts

- Can be implemented differently by each compiler
- User application responsible for knowing specifics if it cares
C Specification
Consumers

- C language specification [formal - threads]
- C compiler
- Pthreads specification [informal + threads]
- User program
Pthreads Specification

• Informal, i.e. English

• Justification: Programmers cannot read formal specifications

• If it was formal, it would extend the C language specification

• If C and Pthreads had formal proofs of metatheory, paper would not exist
Pthreads Specification

• Access to memory locations shared by multiple threads must be restricted via synchronization primitives
  • `pthread_mutex_lock/unlock`

• C Compiler logically treats primitives like function calls accessing global data

• Prevents hardware instruction reordering via additional instructions (memory barriers)
Pthreads Specification

- C Compiler logically treats primitives like function calls accessing global data

\[ f() \quad is \ like \quad pthread_mutex_lock() \]

Initial state
\[ x = 0 \]
\[ f() \quad \{ \quad x++ \quad \} \]

Prevented Incorrect reordering
\[ x = 1 \]
\[ f() \quad [\quad x==2 \quad] \neq \quad f() \quad [\quad x==1 \quad] \]
\[ x = 1 \]
Pthreads Specification

• Problems
  • Incorrect / not safe
  • Not always performant due to lack of non-locking primitives
Memory Model

• Defines which variable assignments are visible to other threads

• More practically

• Code transformations to consider when determining if a variable is shared between threads

• If a variable is shared after transformations, then lock
Sequential Consistency Memory Model
Initial state

\[ x = y = 0 \]

Thread 1

\[ x = 1 \]
\[ r1 = y \]

Thread 2

\[ y = 1 \]
\[ r2 = x \]
Interleaving

**Thread 1**
\[
x = 1 \\
r1 = y
\]

**Thread 2**
\[
y = 1 \\
r2 = x
\]
Interleaving

Thread 1
\[ x = 1 \]
\[ r1 = y \]

Thread 2
\[ y = 1 \]
\[ r2 = x \]
Sequential Consistency

- Intuitive
- Not used much in practice
- Hard to optimize
  - Reordering of operations by compiler
  - Reordering of operations by hardware
Weaker Memory Model
Thread-sequential Reordering

Thread 1
\[ x = 1 \]
\[ r_1 = y \]

Thread 2
\[ y = 1 \]
\[ r_2 = x \]

Thread 1
\[ r_1 = y \]
\[ x = 1 \]

Thread 2
\[ r_2 = x \]
\[ y = 1 \]
# Reordering & Interleaving

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 1</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1 )</td>
<td>( r_1 = y )</td>
<td>( r_1 = y[0] )</td>
</tr>
<tr>
<td>( r_1 = y )</td>
<td>( x = 1 )</td>
<td>( r_2 = x[0] )</td>
</tr>
<tr>
<td>( y = 1 )</td>
<td>( r_2 = x )</td>
<td>( x = 1 )</td>
</tr>
<tr>
<td>( r_2 = x )</td>
<td>( y = 1 )</td>
<td>( y = 1 )</td>
</tr>
</tbody>
</table>
3 Pthreads Bugs
1. Concurrent Modification

- Pthreads mandates protecting shared variables
- What variables should be protected is determined by memory model, which is unspecified
- If you assume sequential consistency, you do not need a lock
- However, a bug occurs because weak consistency speculative store transformations are performed
Interleaving Sequential Consistency

**Initial state**
\[ x = y = 0 \]

**Thread 1**
\[
\text{if} \ (x == 1) \\
\ y++
\]

**Thread 2**
\[
\text{if} \ (y == 1) \\
\ x++
\]
Interleaving Sequential Consistency

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

Thread 2
if (y == 1)
    x++
Speculative Store & Interleaving
Weaker Consistency

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

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

Thread 1
y++ [y==1]
if (x != 1)
y-- [y==0]

Thread 2
x++ [x==1]
if (y != 1)
x-- [x==0]
Speculative Store & Interleaving
Weaker Consistency

Thread 1
y++ \ [y==1]  
if (x != 1)  
  y-- \ [y==0]  

Thread 2
x++ \ [x==1]  
if (y != 1)  
  x-- \ [x==0]  

y++ \ [y==1]  
x++ \ [x==1]  
if (x != 1)  
  // noop \ [y==1]  
if (y != 1)  
  // noop \ [x==1]  

2. Rewriting Adjacent Data

• Previous “speculative store” optimization fishy because we introduce operations on a variable in the source

• Current bug happens even though the optimization only adds operations to variables not in the original source

• Occurs due to adjacent data transformations when partially updating C structs
Adjacency Transformation

Initial state
struct { int a:0; int b:0 } x

Thread 1

\[
\begin{align*}
x.a &= 42 \\
tmp &= x \text{ // Read both fields into} \\
& \quad \text{// 32-bit variable.} \\
tmp &= \sim0x1fff \text{ // Mask off old a.} \\
tmp &= 42 \\
x &= tmp \text{ // Overwrite all of x.}
\end{align*}
\]
Adjacency Transformation

locking a but not b as it is not shared between threads

Thread 1

x.a = 1

Thread 2

x.a = 2
x.b = 3

Thread 1

x.a = 1
x.b = 0

Thread 2

x.a = 2
x.b = 3
## Adjacency Transformation & Interleaving

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x.a = 1$</td>
<td>$x.a = 1$</td>
<td>$x.a = 1$</td>
</tr>
<tr>
<td>$x.b = 3$</td>
<td>$x.b = 0$</td>
<td>$x.a = 2$</td>
</tr>
<tr>
<td>$x.a = 2$</td>
<td>$x.a = 2$</td>
<td>$x.b = 3$</td>
</tr>
<tr>
<td></td>
<td>$x.b = 3$</td>
<td>$x.b = 0$</td>
</tr>
</tbody>
</table>
3. Register Promotion

• Would like to promote a variable to a register sometimes for speedup

• Example is promoting a variable in a loop to a register because profiling determined that the loop usually runs without threads
\(mt\) is whether or not we are using threads

for (...)  
...  
if (mt)  
    lock()  
    x = ...[x]...  
if (mt)  
    unlock()
r = x
for (...
...
if (mt)
  x = r
  lock()
  r = x
  r = ...[r]...
if (mt)
  x = r
  unlock()
  r = x
x = r
Initial state

\[ x = 0 \]
\[ f() \{ \ x++ \ \} \]

Thread 1

for(...)
\[ f() \]
\[ x = 2*x \]

Thread 1

\[ r = 0 \]
for(...)
\[ x = r \]
\[ f() \]
\[ r = x \]
\[ r = 2*r \]

\textit{now we have an } x \textit{ read “without a lock”}

\textit{threads could interleave such lockless code}
Poor Performance

• Locking is useful for most concurrent programs

• Some concurrent programs can be more efficiently expressed by directly using hardware primitives that locks implicitly use
Sieve of Eratosthenes

```plaintext
for (my_prime = 2
 ;my_prime < 10,000
 ;my_prime++)
 if (!get(my_prime))
   for (multiple = my_prime
 ;multiple < 100,000,000
 ;multiple += my_prime)
     if (!get(multiple))
       set(multiple)
```
# Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to any given limit. It works by iteratively marking the multiples of each prime number starting from 2.

## Algorithm

1. **Outer Loop:** Iterate over numbers starting from 2 up to the limit.
2. **Inner Loop:** For each number in the outer loop, if it has not been marked, place it in the inner loop and mark all its multiples as composite.

### Table Example

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Outer loop: 2

Inner loop: 4

```
2
6
8
```

```
3
9
12
```

```
4
12
16
```
Sieve execution time for bit array (secs)
Sieve execution time for bit array (secs)

• The mutex is around segments of the array
• Spin lock performs worse when processors are in heavy use
• The lock-free (atomic instruction) version does not have this problem
Conclusion

- Threading should be a part of a formal language specification, rather than a user library
- Failing to do so results in incorrect compilation and the inability to express lock-free algorithms, resulting in poor performance