“THREADS CANNOT BE IMPLEMENTED AS A LIBRARY”
HANS-J. BOEHM, HP LABS

Presented by
Seema Saijpaul
CS-510
Premise

• Multithreaded programs are often written in languages that are not designed with threading primitives.

• Thread support in these languages are added through thread libraries (Eg: Pthread for C/C++).

• Authors highlight that without a compiler designed explicitly to handle threads/concurrency through language semantics, correctness of code can’t be guaranteed.
Sequential Consistency

- Consider the code: initially with \( x = 0 \) and \( y = 0 \);

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1; )</td>
<td>( y = 1; )</td>
</tr>
<tr>
<td>( r1 = y; )</td>
<td>( r2 = x; )</td>
</tr>
</tbody>
</table>

- Sequential consistency: Assignment of either \( x=1 \) or \( y=1 \) must happen first before assigning \( r1/r2 \) to \( x/y \).
  - So, value of either \( r1 \) or \( r2 \) should be \( >0 \);
Sequential consistency (cont…)

- This memory model is very restrictive and not true for many modern architectures
  - Compilers can reorder memory operations if they don’t violate intra-thread dependencies;
  - Hardware can reorder memory operations as well to improve performance.

- \( r1 \) and \( r2 \) can both be equal to 0

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r1 = y; )</td>
<td>( r2 = x; )</td>
</tr>
<tr>
<td>( x = 1; )</td>
<td>( y = 1; )</td>
</tr>
</tbody>
</table>
Concurrency and Language

- Languages such as C/C++ don’t include threads as part of the language specifications.
  - Hence, No concurrency support.

- Compiler reorders memory instructions without knowing concurrency impact to improve performance which might lead to race conditions.
Pthread library

• C/C++ languages support thread using Pthread library.

• But Pthread standard doesn’t specify how language should handle concurrency and delegates to the developer to ensure that all shared memory location access to be synchronized using locks.
  • pthread_mutex_lock – to acquire a lock
  • pthread_mutex_unlock – to release a lock
Programmer’s dilemma

• The onus is now on application developer to ensure correctness of the software by applying locks at appropriate places.

• How to determine when the memory reordering will occur leading to race conditions?
  • This requires formal memory model which the C/C++ language doesn’t provide.
  • Thus making this problem circular.
Disabling Optimizations

- In practice, C/C++ implementations supporting Pthread
  - Use locking functions that include hardware instructions ("memory barriers") that prevent hardware reordering of memory operations around the call.
  - Disable compiler optimization to prevent compiler from reordering memory operations around locking calls.

This works but NOT always! Race conditions are still possible.
Correctness Issue 1: Concurrent modification

- **Scenario #1:** If \( x = 0, \ y = 0 \) initially,

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>If ((x == 1) \text{ ++y;})</td>
<td>If ((y==1) \text{ ++x;})</td>
</tr>
</tbody>
</table>

  Expected final value: \( x = 0 \) and \( y = 0 \) as well.

- **Scenario #2:** If compiler reorders memory instructions

<table>
<thead>
<tr>
<th>Thread #1</th>
<th>Thread #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{++y;} \text{ If } (x != 1) { --y;}</td>
<td>\text{++x;} \text{ If } (y!=1) { --x;}</td>
</tr>
</tbody>
</table>

  Here \( x \) and \( y \) values can’t be guaranteed to be 0 in concurrent environment.

Compiler assumes no concurrency leading to race conditions
Solution to the issue

• Programmer’s action:
  • Programmer can avoid race condition by placing locks that will prevent the compiler from optimizing the code

• Compiler’s action:
  • Compiler can avoid reordering the code if it’s aware of the concurrency issues that can arise due to threads.

• In both the cases, change in the language specification is needed
  • i.e Memory model and concurrency semantics addition
Correctness issue 2: Adjacent Data overwrite

Consider the code:
```c
struct {
    int a:17;
    int b:15;
} x;

{  // Store it to temp variable
    tmp = x;
    tmp &= ~0x1ffff; // Mask the lower 17 bits
    tmp |= 42; // Adding the value of 42 to variable
    x = tmp; // updating the entire 32-bits into x
}
```

Let's update `x.a = 42;`

At this instant, another thread sets `x.b = 391`

Original `x.b` is lost
Correctness issue 2: Adjacent Data overwrite (cont.)

• Even if x.a and x.b are protected by separate locks, the overwrite will happen
  • Programmer has to acquire both locks for x.a and x.b before update
  • How can the programmer know about this?

• Language specification needs to be define clearly when adjacent bit-fields can be overwritten

• How many adjacent bit-fields will be copied depends if architecture allows bit, byte or word sized copies.
  • Optimizing for such architecture will make the SW non-portable.
Correctness issue 3: Register promotion

- Compiler optimizations can introduce race conditions that were not in source

```c
r = x;  // extra reads/ writes introduced
for (...) {
    if (mt) {
        x = r;
        pthread_mutex_lock();
        r= x;
    }
    r = ... r ...;
    if(mt){
        x = r;
        pthread_mutex_unlock();
        r=x;
    }
}
x = r;
```
Correctness issue 3: Register promotion

- Compiler promotes the frequently updated variable within the loop to a register for better performance.

- The extra read and write operations on the variable are introduced when the lock is not held.
  - This can introduces race condition despite not being in source code.

- Compiler being cognizant of threads can avoid this register promotion optimization thereby preventing race.
Performance

• Pthreads standard advocate concurrent access to shared variable be protected by using mutual exclusion.
  • Mutual exclusion locks utilize hardware atomic instructions that are much more expensive than normal operations.
  • This has an impact on the performance of the concurrent SW.

• Author suggests C/C++ language specification should support lock-free programming methods as well.
Performance comparison

Graphs show that performance of different concurrency techniques for Sieve of Eratosthenes algorithm.

- Non-blocking methods using atomic operations provide the biggest performance boost.
Conclusion

• Compilers need to be aware of threads for correctness of concurrent programs
  • Threads as libraries cannot guarantee correctness.
  • Memory model should be part of language specification.

• Potential opportunity to increase performance on multiprocessor system using lock-free methods.

• Joint effort from Boehm et. al to address these issues in C++ standard.
Attributions

• Prof. Jonathan Walpole, from CS510-Spring 2010