# Threads Cannot Be Implemented As A Library

Hans-J. Boehm

Presented by Burcea Mihai

# Agenda

- Identifying and Understanding the problem
- The *Pthreads* Approach to Concurrency
- Correctness Issues
- Performance Issues
- Conclusion/discussion

# Identifying the Problem

- Most multithreaded programs use a shared memory model
- For C/C++, multithreading is not part of the language specification
- Instead, thread support is being provided through the means of libraries
- *Pthreads* – most popular threads library

# Identifying the Problem

- Claims:
  - These environments are underspecified
  - Correctness of written programs derives from implementations, not from the standards/specs
  - However, the problem is in the compiler, and the language specification, not in the library or the thread library specification
  - Also, library-based approaches may exhibit suboptimal performance in certain cases

# *Pthreads* Approach to Concurrency

- Traditional memory model:

  Thread 1: x = 1; r1 = y;

  Thread 2: y = 1; r2 = x;

- Upon completion, either r1 or r2 *must* be 1
- This model is called *sequential consistency*
- However, in most realistic programming languages with true concurrency support, r1 = r2 = 0 is acceptable

# *Pthreads* Approach to Concurrency

- Two reasons for this:
  - Instruction reordering by (non-thread-aware) compiler for better performance
    - Doing so is not incorrect in the context of single threaded execution
  - Instruction reordering by the hardware
    - E.g., x86 may reorder a store followed by a load
- This is a weaker memory model, and both Java and *Pthreads* allow for this

# *Pthreads* Approach to Concurrency

- In practice, C/C++ implementations do this:
  - Synchronization functions like *pthread_mutex_lock* include hardware instructions that prevent hardware reordering of memory operations around the call
  - To prevent the compiler from reordering them, such functions are treated as opaque functions (can potentially modify any global variable), and thus memory operations are not moved around the call

# *Pthreads* Approach to Concurrency

- This works *most* of the time
- Not always, because it does not define precisely when a data race may occur, or when the compiler may introduce one
- Another problem: this solution sometimes excludes the best performing algorithmic solutions; therefore, many systems violate these rules intentionally

# Correctness Issues: Concurrent Modification

- *Pthreads* prohibits races (access to a shared variable while another thread is modifying it)
- But the existence of a race is defined by the semantics of the language…
- Which in turn requires the existence of a properly defined memory model !

# Correctness Issues: Concurrent Modification

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

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

- Under sequential consistency model: there is no race, and the only valid outcome is x = y = 0
- What if the compiler optimizes these statements ?…

# Correctness Issues: Concurrent Modification

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

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

- This is a race, hence semantics of this programs is undefined
- x = y = 1 is a perfectly possible outcome
- Reason? Compiler is unaware of threads, and its optimizations are perfectly legal when *only* considering the sequential consistency model

# Correctness Issues: Rewriting of Adjacent Data

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

- The assignment x.a = 42; may be implemented like this:

```
{
    tmp = x; //read both fields into 32-bit var.
    tmp &= ~0x1ffff; //mask off old a.
    tmp |= 42;
    x = tmp; //overwrite all of x.
}
```

# Correctness Issues: Rewriting of Adjacent Data

- This is ok for sequential code
- But a race appears if a concurrent update to x.b occurs between ‘*tmp = x*’ and ‘*x = tmp*’
- Even though the two threads operate on distinct fields, the update may be lost
- Same problem for other cases…

# Correctness Issues: Rewriting of Adjacent Data

- 64-bit machine, compiler knows that x is 64-bit aligned

  struct {char a;char b;char c;char d;

  char e;char f;char g;char h;} x;

- Assume sequence of assignments:

  x.b = ‘b’; x.c = ‘c’; x.d = ‘d’; x.e = ‘e’;

  x.f = ‘f’; x.g = ‘g’; x.h = ‘h’;

# Correctness Issues: Rewriting of Adjacent Data

- The compiler might compile this into the more efficient

  x = ‘hgfedcb\0’ | x.a ;

- This introduces a race with a concurrent assignment to x.a, even though the two threads access disjoint sets of fields

# Correctness Issues: Rewriting of Adjacent Data

- This may even happen for adjacent global variables outside a *struct* declaration
- Linkers commonly reorder globals, therefore an update to a global variable may potentially read/write any other global variable

# Correctness Issues: Register Promotion

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

- The lock is acquired conditionally, depending on whether a second thread has been started inside the process

# Correctness Issues: Register Promotion

- Compiler determines conditionals are usually not taken, so it promotes x to a register in the loop
- It treats the two *pthread* synchronization functions as opaque function calls
- Hence, the code might look like:

# Correctness Issues: Register Promotion

```
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;
```

# Correctness Issues: Register Promotion

- The *pthreads* standard requires that memory be synchronized with the logical program state at the two sync function calls
- This is satisfied by the above code
- However, now there are reads and writes of x while the lock is not held
- So code is broken and incorrect, while satisfying the (insufficient) *pthreads* specs

# Performance

- *Pthreads* imposes concurrent access to shared variables through sync. library calls
- Hardware atomic instrs. are very expensive (> 100 register-to-register instrs.)
  - – x86: atomic update of memory: 100+ cycles
- *Pthreads* primitives built on top of these are even more expensive

# Performance

- For better performance: use lock-free and wait-free programming techniques and benefit from data races
- Example: Sieve of Eratosthenes for 100M elements (extracted from garbage collection code)
- Array initialized to *false*, *get(i)* is *A[i]* and *set(i)* is *A[i]=true*

# Performance

```
for (my_prime = start;my_prime < 10000;
  ++my_prime)
  if (!get(my_prime)) {
      for (multiple = my_prime;multiple <
        100000000;multiple += my_prime)
          if (!get(multiple)) set(multiple);
  }
```

# Performance

- Primes below 10k are not computed
- On completion, *get(i)* is *false* iff *i* is prime
- But this works (correctly) for multiple threads all accessing the same array, too !
- Because:
  - For a thread not to invoke *set* on all multiples of some *j*, *get(j)* must have returned *true*
  - But then some other thread must have called *set(j)*, and, consequently, on all multiples of *j*

# Performance

- 4-way multiprocessor (1GHz Itanium 2), Debian Linux, gcc3.3
- 4 implementations: *pthread* mutex sync, spin-locks, volatile accesses without other synchronization, and no synchronization at all
- Only first 2 are compatible with *pthread* rules

# Itanium2 performance on byte array

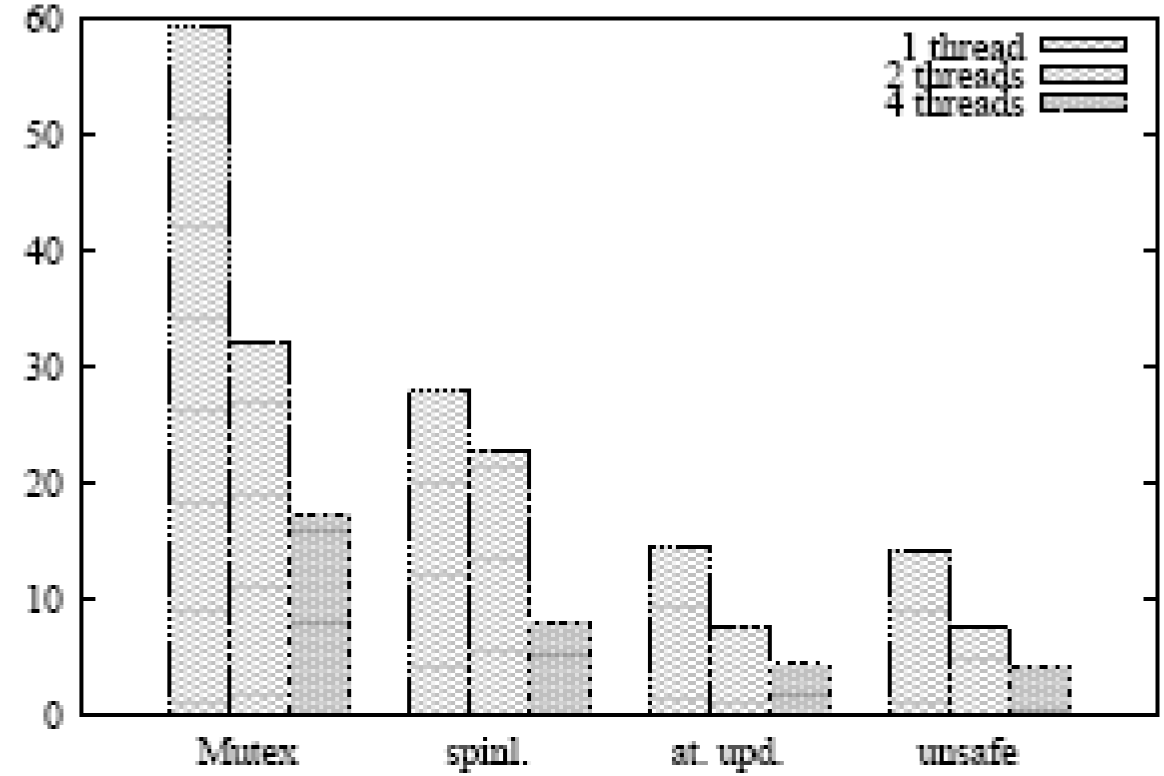


**Figure 1.** Sieve execution time for byte array (secs)

# Itanium2 performance on bit array

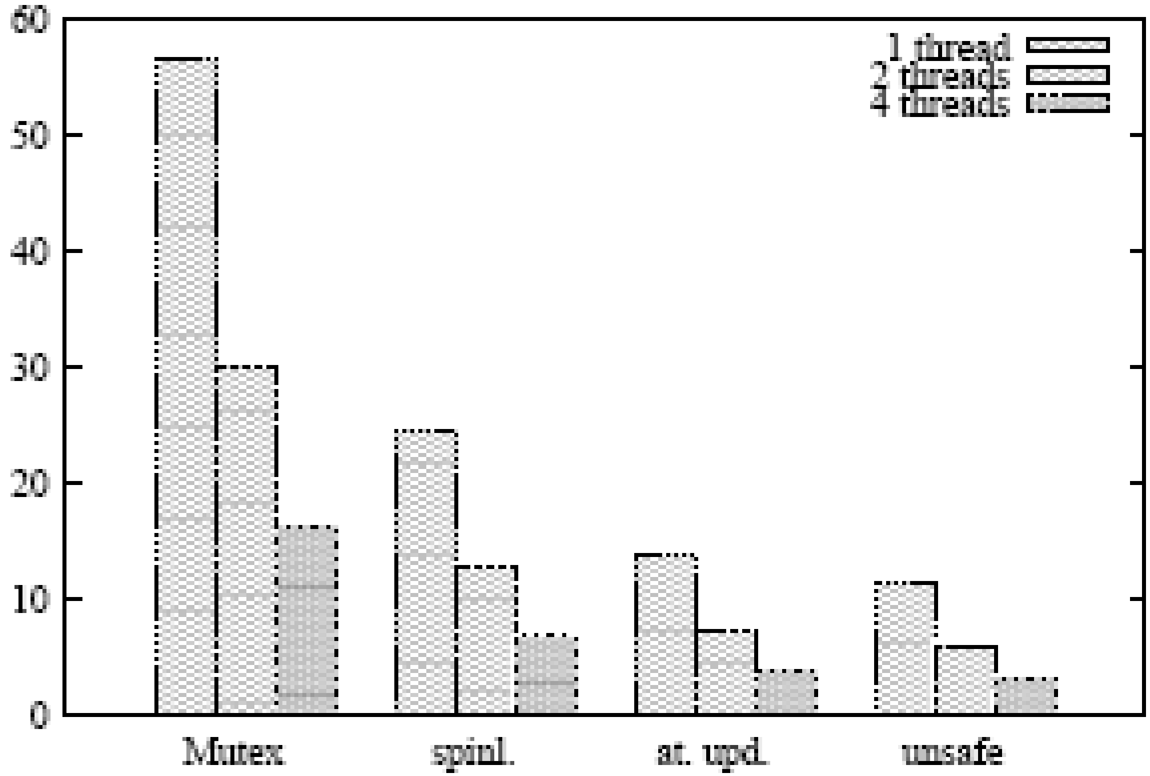


**Figure 2.** Sieve execution time for bit array (secs)

# HT P4 performance

- Hyperthreaded Pentium 4 (2 GHz, 2 CPUs with 2 threads each), Fedora Core Linux
- Higher sync costs, hence we see even higher benefits over the the fully synchronized versions
- Here the single-threaded version appears optimal (most likely because it already saturates the memory system)

# HT P4 performance on byte array

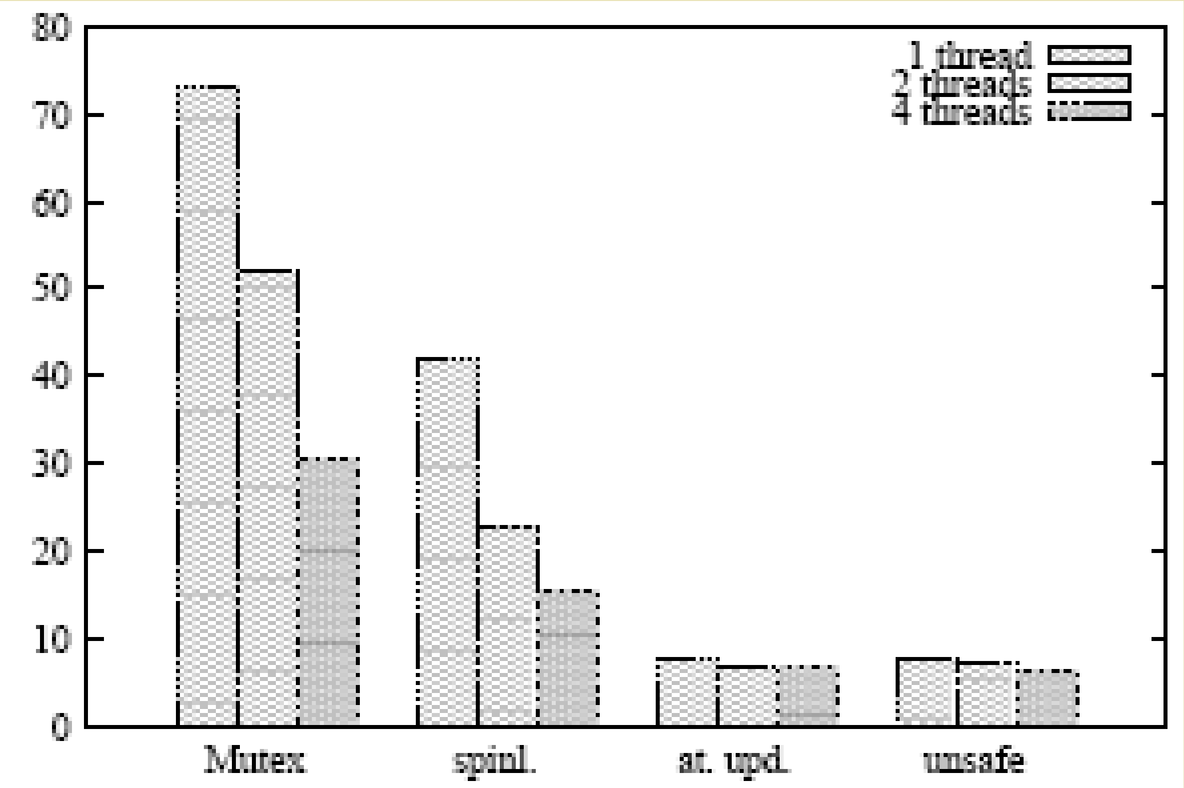


**Figure 3.** HT P4 execution time for byte array (secs)

# Heap tracing of 200 MB on P4

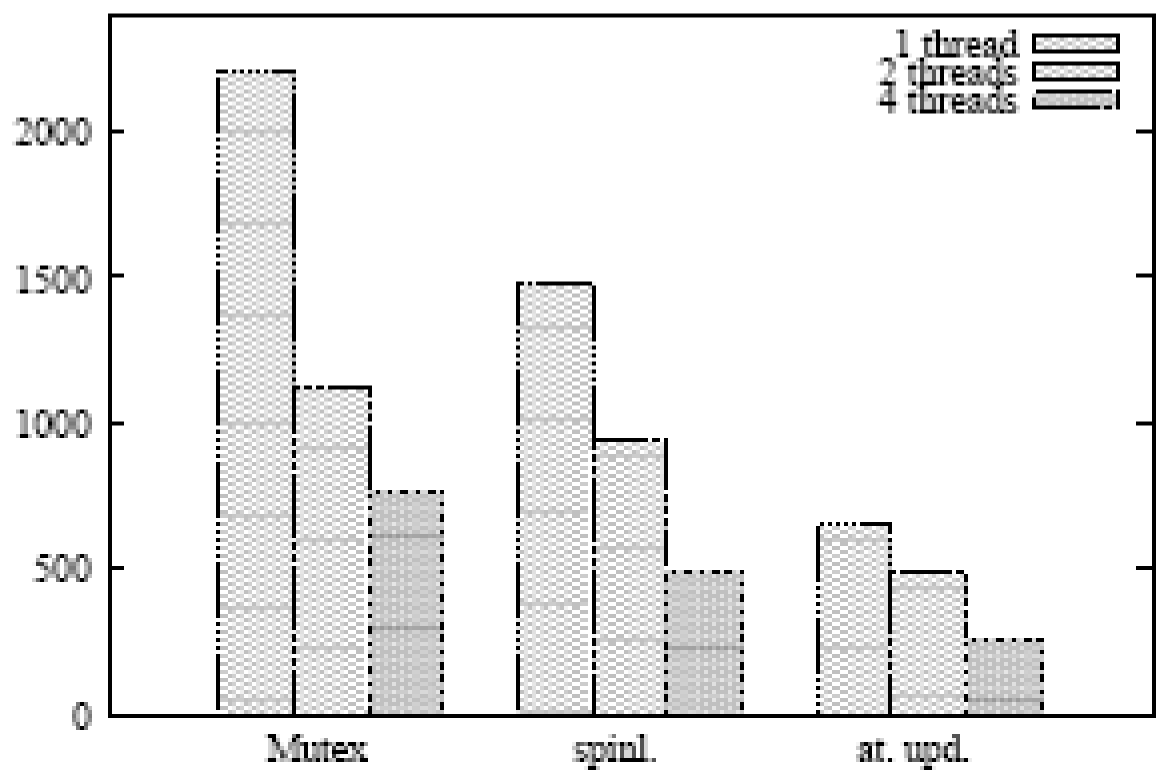


Figure 4. HT P4 time for tracing 200 MB (msecs)

# Consequences of allowing data races

```
x = 1;
pthread_mutex_lock
   (lock);
y = 1;
pthread_mutex_unlock
   (lock);
```

```
pthread_mutex_lock(lock);
y = 1;
x = 1;
pthread_mutex_unlock
         (lock);
```

The transformation on the right may have better performance, even though it contradicts the *pthreads* specs

# Conclusions

- Current state of things may lead to
  - Non-portable code
  - Broken code
  - Suboptimal performance
- Solutions: adopt a proper memory model, similar to Java’s, but more performance-oriented

# Conclusions

- Don’t fully define the semantics of all data races (some may be desirable)
  - E.g. restrict it to *volatile* accesses, or shared variable access through certain library calls
- Don’t prohibit reordering volatile store followed by volatile load
- Account for potential races caused by reordering in the case of bit-fields