CSE 401 – Compilers

Threads and Memory Models

Guest Lecture by Dan Grossman

Winter 2010, Last Day of Class

References

- *Threads Cannot Be Implemented as a Library*
  Boehm, PLDI 2005
- *Foundations of the C++ Concurrency Memory Model*
  Boehm and Adve, PLDI 2008
- *The Java Memory Model*
  Manson, Pugh, and Adve, POPL 2005

Credits: Earlier versions of lecture by
Vijay Menon, CSE 501, Sp09
Hal Perkins, CSE P501, Au09

Threads and shared memory

- Multithreading lets multiple threads run concurrently
  - Each thread has its own local variables (stack and registers), but…
  - All threads share one memory
    - globals / statics + heap objects
  - Use memory to communicate ☺ or interfere ☹
- Becoming more common to exploit multicore hardware
- Basic use / issues: CSE303, CSE378, CSE451
  - New: CSE332, CSE333, maybe CSE331

Naïve view

The following *almost* works

1. Define your programming language "as usual"
   - Don't think about > 1 thread
2. Compile the code like you've learned all quarter
   - Don't think about > 1 thread
3. Provide a run-time library that provides threading
   - Create thread
   - Create/acquire/release *mutual-exclusion locks*
   - Etc.
4. Profit

This lecture in one slide

The naïve approach, followed for decades, is **fatally flawed**

- Compiler must know threads & shared-memory exist
  - Else it may perform **incorrect optimizations**
- Programmer must know threads & shared-memory exist
  - The natural definition ("**sequential consistency**") of how shared-memory works ("the memory model") is **not tractably implementable by compilers or hardware**
  - So we have less-natural weaker definitions to make language implementation easier. Usually defined so that:
    - **If** programmers avoid data races **then** they can ignore this
    - **Most** compiler optimizations remain legal

Safety of optimization

The standard rule for optimization:

If, in some program context, the result of evaluating e1 cannot be distinguished from the result of evaluating e2, the compiler can substitute e2 for e1 in that context

Now: Three gotchas that arise only with multiple threads and shared memory

- Examples use global variables to keep them short; same issues arise with shared objects in the heap
- Examples are **illegal optimizations** in, e.g., Java

Gotcha #1: Speculation

(Probably the least common / well-motivated, but the easiest to understand)

```
// x and y are globals, initially 0

void foo() {
  ++x;
  if(y==1)
    ++x;
}
```

Gotcha #1: Speculation

```
// x and y are globals, initially 0

void foo() {  optimized     void foo() {
  ++x;        =========>      x += 2;
  if(y==1)                    if(y!=1)
    ++x;                        --x;
}                           }
```

Before optimization

```
// x and y are globals, initially 0
```

Thread 1

```
void foo() {
  ++x;
  if(y==1)
    ++x;
}
```

Thread 2

```
void bar() {
  if(x==2)
    commence evil();
}
```

After optimization

```
// x and y are globals, initially 0

Thread 1              Thread 2

void foo() {          void bar() {
  x += 2;               if(x==2)
  if(y!=1)                commence_evil();
    --x;              }
}
```

Recap

So our compiler made a change that:

- Is legal for all single-threaded programs
- Caused execution to “make up” a new value for **x**

So either:

- Our compiler must not do this (thread-aware)
- Or we must change our language definition to allow this (bad idea in this example)

Gotcha #2: Register promotion

```
// x is global, initially 0

void foo(int* a, int n) {
  for(int i=0; i<n; ++i)
    x += a[i];
}
```

optimized

```
========>
```

```
void foo(int* a, int n) {
  int reg = x;
  for(int i=0; i<n; ++i)
    reg += a[i];
  x = reg;
}
```

Before optimization

```
// x is global, initially 0

// Thread 1                      // Thread 2

void foo(int* a, int n) {        void bar() {
  for (int i = 0; i < n; ++i)      x = 10;
    x += a[i];                     ...
}                                }
```

What happens when n == 0?

After optimization

```
// x is global, initially 0

// Thread 1                          // Thread 2

void foo(int* a, int n) {            void bar() {
  int reg = x;                         x = 10;
  for (int i = 0; i < n; ++i)          ...
    reg += a[i];                     }
  x = reg;
}
```

What happens (sometimes) when n == 0?

Recap

In executions where **`n==0`**, the compiler optimization can “lose an update”

- Original code: **`x==10`** is guaranteed for code after both threads finish
- Optimized code: **new write** of **`x = 0`** creates new possible result

Gotcha #3: Adjacent data

```
char arr[4];

void foo() {
  arr[0] = (char)0;
  arr[1] = (char)0;
  arr[2] = (char)0;
}
```

Natural assembly for body:

```
movb $0, _arr
movb $0, _arr+1
movb $0, _arr+2
```

Assembly with one store:

```
movl _arr, %eax
andl $0x000000FF, %eax
movl %eax, _arr
```

Before optimization

```
char arr[4];

// Thread 1:
movb $0, _arr
movb $0, _arr+1
movb $0, _arr+2
```

```
// Thread 2
// arr[3] = ‘a’;
movb $98, _arr+3
```

After optimization

```
char arr[4];

// Thread 1:
movl _arr, %eax
andl $0x000000FF, %eax
movl %eax, _arr
```

```
// Thread 2
// arr[3] = ‘a’;
movb $98, _arr+3
```

Recap

The clever compiler is adding the assignment "`arr[3]=arr[3];`"

- That's fine in single-threaded code

In practice, this is a problem if:

- Your architecture doesn't have byte-stores
  - Leave space between string characters??
- You have bit-fields in C (and no bit-stores)
  - C++ specifically allows the "clever" code because there is no other way (so programmer must avoid simultaneous write to bit-fields in same struct)

Where are we

- So far have emphasized that the **compiler** must limit itself in order to be correct in the presence of threads
  - This is 401 after all
- You should also understand that the **programmer** must accept unintuitive language definitions
  - Otherwise efficient compiler/hardware too difficult
  - Simple answer: Never write code with a data race
  - Must discuss **memory-consistency models**

Dekker's example

- Initially, **`x==0 && y==0`**

```
Thread 1                 Thread 2
 x = 1;     (a)           y = 1;     (c)
 r1 = y;    (b)           r2 = x;    (d)
```

- What are possible executions?

Dekker's example

- Initially, **`x==0 && y==0`**

```
Thread 1              Thread 2
 x = 1;    (a)         y = 1;    (c)
 r1 = y;   (b)         r2 = x;   (d)
```

- What are possible executions?
- Consider interleavings of thread 1 & 2:
  - abcd, acbd, acdb, cdab, cadb, cabd
  - (24 permutations, but need a before b and c before d)

Dekker's example

- Initially, **`x==0 && y==0`**

```
    Thread 1            Thread 2
     x = 1;              y = 1;
     r1 = y;             r2 = x;
```

- Can **`r1 == 0 &&  r2 == 0`** ?
  - No interleaving gives this results, but…
  - Most hardware will allow it
    - Store buffers; see CSE471
  - Most compilers will allow it
    - Why…

Compiler reordering

- Almost every compiler optimization has the implicit effect of reordering reads and writes!
  - Obvious example: Instruction scheduling
  - Less-obvious example: Common-subexpression elimination

```
x=a+b;
y=a;
z=a+b; //optimize to z=x
```

  Replacing with **`z=x`** *has the effect* of

  moving the store to **`z`** to before the store to **`y`**!

    - **`y`** could see a later write to **`a`** by another thread than **`z`** sees

Sequential consistency

- The interleaving model is called sequential consistency and was defined in 1979 by Lamport:

  "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
- But no “real” hardware or compiler implements it
- So we have to tell programmers what they ***can*** assume

Refined notion

- Guarantee sequential consistency only for ***correctly synchronized*** programs (Adve)
  - Give the programmer rules to follow
  - Promise interleaving semantics ***if*** rules are obeyed
- Correctly synchronized
  - Must be intuitive to programmer
  - Must not be restrictive for compiler/hardware

Data races

- Two operations *conflict* if they both access a memory location and one is a write
- A execution contains a *data race* if two adjacent operations from two different threads conflict
  - `x = 1; y = 1; r1 = y; r2 = x;`
- A program is data-race-free if no sequentially consistent execution (i.e., interleaving) has a data race

Correct synchronization

- We call a program correctly synchronized if it is data race free

- Basic contract – "The Grand Compromise":
  - *If* programmers write data-race-free programs, implementers will provide sequentially consistent semantics
  - This is the fundamental property of the Java and C++ memory models

How do we avoid races?

- Mutual exclusion:
  - Thread acquires lock before accessing a shared variable
  - Locks exist to avoid races

```
Thread 1                      Thread 2
  lock (mutex);                 lock (mutex);
  tmp1 = x;                     tmp3 = x;
  tmp2 = tmp1 + 1;              tmp4 = tmp3 + 1;
  x = tmp2                      x = tmp4
  unlock (mutex);               unlock (mutex);
```

- Java’s volatile variables (atomics in C++)
  - Data races allowed; compiler can’t reorder

What this means for compilers

- In the absence of synchronization, compilers may *almost* operate as if programs were single-threaded
- Compilers must respect ordering due to synchronization (locks, volatiles, etc.)
  - Even if “hidden” inside a function/method call
- Compilers must not introduce data races into correctly synchronized code
  - This is why Gotchas #2 and #3 are illegal for compilers!
  - They add writes that race with the program!

What happens on a race?

- In C++, undefined semantics

```
Thread 1                   Thread 2
 x = 1;    (a)              y = 1;    (c)
 r1 = y;   (b)              r2 = x;   (d)
```

- Valid results:
  - `r1 == 0` and `r2 == 0`
  - `r1 == 0` and `r2 == 42`
  - `system(rm –rf /*);`
- No such thing as a benign data race in C++!
  - Hence Gotcha #1 is legal in C++ because the original program had a data race

Type-safety issues

- In Java, data races cannot violate type safety
  - Java promises a measure of security
  - Cannot allow data races to be used on purpose by untrusted code to open / exploit holes
  - Java memory model must provide some guarantees even in the presence of races
    - Gotcha #1 is illegal in Java; cannot make up values

Java reality

- The actual “memory model” (what can and can’t happen with reads/writes) is very complicated
  - Took years by brilliant people and still had problems
- **Programmers** willing to *avoid data races* do not need to understand the definition
  - There is a theorem about the definition that all data-race free programs behave as in the interleaving semantics
- But **compiler writers** must *avoid gotchas*
  - Very roughly speaking, don’t make up values or introduce data races

This lecture in one slide

The naïve approach, followed for decades, is **fatally flawed**

- Compiler must know threads & shared-memory exist
  - Else it may perform **incorrect optimizations**
- Programmer must know threads & shared-memory exist
  - The natural definition ("**sequential consistency**") of how shared-memory works ("the memory model") is **not tractably implementable by compilers or hardware**
  - So we have less-natural weaker definitions to make language implementation easier. Usually defined so that:
    - **If** programmers avoid data races **then** they can ignore this
    - **Most** compiler optimizations remain legal