Concurrency and Optimization

Evan Herbst

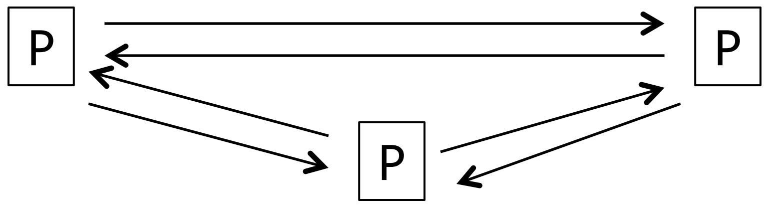
adapted from Hal Perkins' fall '11 CSEP501 slides

Terminology

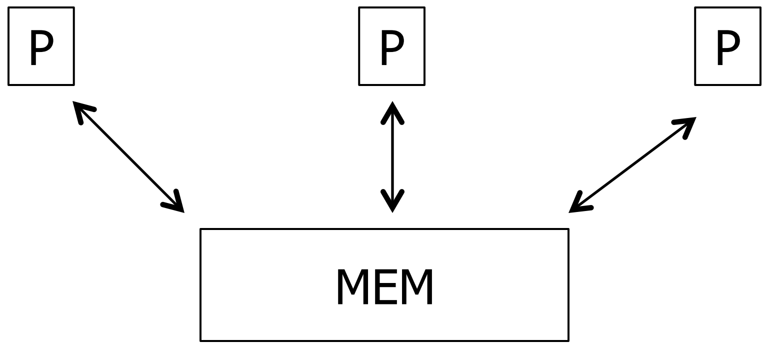
- Parallelism: a property of a *computation* that lets us execute multiple pieces of it at once
- Concurrency: the property of an *execution* actually taking advantage of parallelism

Terminology

- Message-passing concurrency



- Shared-memory concurrency



Terminology

- Thread (“thread of execution”): state needed for running a program assuming the code is mostly self-contained; generally means PC, stack, registers
- A process can be thought of as a thread plus
  - Contents of memory
  - OS resources (eg file descriptors, sockets)

Safety of optimization

- A standard constraint / definition:
  - *If, in their actual program context, the result of evaluating e' cannot be distinguished from the result of evaluating e, the compiler can substitute e' for e*
- What does this mean in a multi-threaded setting?

Register promotion

```
// x is global, initially 0

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

Register promotion

```
// x is global, initially 0

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

$\longrightarrow$

```
// Optimized

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

Before optimization

```
// x is global, initially 0

// Thread 1

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

```
// Thread 2

void bar() {
  x = 10;
  …
}
```

What happens when n == 0?

After optimization

```
// x is global, initially 0

// Thread 1

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

```
// Thread 2

void bar() {
  x = 10;
  ...
}
```

What happens when n == 0?

What happened?

- In executions where n == 0, the compiler optimization creates a value out of thin air
  - Original code: x == 10 is guaranteed
  - Optimized code: new write of x = 0 (inserted x = reg) creates new result

How did we get here?

- C & C++ originally defined as single-threaded languages
  - Compilers didn't consider threads
  - Threads were provided by external libraries (e.g. pthreads) that defined their own semantics
- This is a broken model!
  - New specs explicitly deal with threads (Boehm, et al)

Dekker’s example

- Initially, x == 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 == 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

Dekker’s example

- Initially, x == y == 0

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

- Can r1 == r2 == 0?
  - No interleaving gives this results, but...
  - Most hardware will allow it (store buffers)
  - Many compilers will allow it (instruction scheduling)

What is a correct execution?

- Simplest notion: *sequential consistency* (Lamport '79)
  "... 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."
- This is essentially the interleaving model
- Too expensive (?)
  - Nobody implements this in practice

Refined notion

- Guarantee sequential consistency only for *correctly synchronized* programs (Adve)
  - Give the programmer rules to follow
  - Give simple semantics when rules are obeyed
- Correctly synchronized
  - Must be intuitive to programmer
  - Must not be restrictive for implementer

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
  - $x = 1;\ \underline{y = 1;\ r1 = y;}\ r2 = x;$
- A program is 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:
  - If programmers write race free programs, implementers will provide sequentially consistent semantics
  - This is the fundamental underpinning for Java, C, and C++ memory models

Another example

- Dekker's example is not race free
- What about: (initially, x == y == 0)

```
Thread 1                  Thread 2
  r1 = x;                   r2 = y;
  if (r1 > 0)               if (r2 > 0)
    y = 1;                    x = 1;
```

How do we avoid races?

- Mutual exclusion:
  - Thread acquires lock before accessing a shared variable:

```
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);
```

  - Locks disallow problematic interleavings

How do we avoid races?

- Volatile variables ('atomic' in C++11):
  - Certain variables are declared with stronger ordering semantics (initially, x and flag are 0):

```
Thread 1                    Thread 2
  x = 1;                      if (flag == 1)
  flag = 1;                     t = x;
```

  - If flag is declared volatile, then write to x cannot be sunk in T1 and read from x cannot be hoisted in T2 by definition
    - Compiler must respect ordering

What does this mean for compilers?

- In the absence of synchronization, compilers may *almost* operate as if programs were single-threaded
- Compilers must respect ordering due to synchronization (and generate necessary hardware instructions)
- Caveat: compiler must not introduce races into correctly synchronized code (e.g. register promotion)

What happens on a race?

- In C++, undefined semantics

```
Thread 1     (x == y == 0)     Thread 2
  x = 1;    (a)                  y = 1;     (c)
  r1 = y;   (b)                  r2 = x;   (d)
```

- Valid results:

What happens on a race?

- In C++, undefined semantics

```
Thread 1     (x == y == 0)     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 = 2
  - "format c:\"
- No such thing as a benign race in C++!

Hard to bound effects

```
unsigned x;

if (x < 3) {
  // x modified by another
  // thread
  switch (x) {
    case 0: ...
    case 1: ...
    case 2: ...
  }
}
```

- Compiler should be able to generate table
  - Assumes x in range after check
  - Async change to x causes arbitrary behavior

A benign data race

- Evan noticed a race similar to this in his code recently

```
vector<bool> flags(50); …fill flags…
vector<bool> flags2(50, false);
start N threads running f(flags, flags2)

void f(vector<bool> v, vector<bool>& v2)
{
        for(int i = 0; i < 50; i++)
                v2[i] = 3 * v[i];
}
```

Java Memory Model

- Data races can't be allowed to violate type safety
- Security is a big deal in java, so need a semantics in the presence of races
- Solution so far: statically defined ordering over runtime operations, including synchronization operations

References

- *Memory Models: A Case for Rethinking Parallel Languages and Hardware*
  Adve and Boehm, CACM Aug. 2010
- *Foundations of the C++ Concurrency Memory Model*
  Boehm and Adve, PLDI 2008
- *Threads Cannot be Implemented as a Library*, Boehm, 2004

- Many slides by Vijay Menon, CSE 501, Sp09