CS162 Week 5

Kyle Dewey

Overview

- Announcements
- Reactive Imperative Programming
- Parallelism
- Software transactional memory

TA Evaluations

unicode_to_ascii.sh

Reactive Imperative Programming Clarification

Tips

- **Never** execute a constraint when you are in atomic mode
- Self-recursive constraints should **never** trigger themselves
- Reactive addresses can be both **added** and **removed** via `newCons`, depending on what gets used in the `newCons`' body
  - If a previously reactive address is not used when executing ~~a constraint~~ `newCons`, the address is no longer reactive

`deletedAddress.not`

Any lingering Reactive Imperative Programming Questions?

Parallelism

Previous Experience

- Anyone taken CS170? CS240A?
- Any pthreads users?
- Threads in any language?
- MPI users?
- Cloud? (AWS / Azure / AppEngine...)
- Comfort with parallel code?

Going back a bit...

- A processor executes instructions
- The faster it can execute instructions, the faster our program runs
- Ideal world: one processor that runs really, really fast

Moore's Law

- The number of transistors per some unit of volume in integrated circuits doubles roughly every two years
- This number is roughly correlated to how fast it runs

In the Beginning: The Land was At Peace

- Processors get faster and faster
- Using clock speed as a poor estimator of actual speed:
  - Pentium: Up to 300 Mhz
  - Pentium II: Up to 450 Mhz
  - Pentium III: Up to 1.4 Ghz
  - Pentium 4: Up to 3.8 Ghz

Then Physics Happened

Heat

- More transistors means more power is needed
- More power means more heat is generated for the same amount of space
- Too much heat and the processor stops working

So Just Cool It

- Again, physics is evil
- Normal heatsinks and fans can only push away heat so quickly
- To get heat away fast enough, you need to start getting drastic
  - Water cooling
  - Peltier (thermoelectric) coolers
  - Liquid nitrogen drip

Even if it Could be Cooled...

- When transistors get too close together, quantum physics kicks in
- Electrons will more or less **teleport** between wires, preventing the processor from working correctly
- Not cool physics. Not cool.

But the Computer Architects had a Plan...

...one that would allow processors to keep getting faster...

“Eh, I give up. Let the software people handle it.”

Multicore is Born

- Put multiple execution units on the same processor
  - Uses transistors more efficiently
- Individual cores are slower, but the summation of cores is faster
  - I.e. 2 cores at 2.4 Ghz is "faster" than a single processor at 3.8 Ghz

Problem

- The software itself needs to be written to use multiple cores
- If it is not written this way, then it will only use a single core
- Nearly all existing software was (and still is) written only to use a single core
  - Oops.

Why it is hard

- People generally do not think in parallel
  - Want to spend more time getting less done poorly? Just multitask.
- Many problems have subproblems that must be done sequentially
  - Known as **sequential dependencies**
  - Often require some sort of communication

In the Code

- With multiple cores, you can execute multiple **threads** in parallel
- Each thread executes its own bit of code
- Typical single-core programs only have a single thread of execution
- One explicitly requests threads and specifies what they should run

Example

```
int x = -1;
```

```
void thread1() {
  if (x == -1) {
    x = 5;
  }
}
```

```
void thread2() {
  if (x == -1) {
    x = 6;
  }
}
```

Race Conditions

- This example still **may** get executed correctly
  - Depends on what gets run when
- This is called a **race condition**
  - One computation "races" another one, and depending on who "wins" you get different results
- IMO: the most difficult bugs to **find** and to fix

Fundamental Problem

- Need to manage shared, mutable state
- Only certain states and certain state transitions are valid
  - In the example, it is valid to go from -1 to 5, or from -1 to 6, but not from 5 to 6 or from 6 to 5
- Need a way of enforcing that we will not derive invalid states or execute invalid state transitions

A Solution: Locks

- Shared state is under a lock
- If you want to modify it, you need to hold a key
- Only one process can hold a key at a time

Example With Locks

```
int x = -1;
```

```
void proc1() {
  lock (x) {
    if (x == -1) {
      x = 5;
    }
  }
}
```

```
void proc2() {
  lock (x) {
    if (x == -1) {
      x = 6;
    }
  }
}
```

Problems With Locks

- Very low-level and error prone
- Can absolutely kill performance
  - Because of locks, the example before is now purely sequential, **with locking overhead**

Deadlock

```
int x = 1;
int y = 2;
```

```
void proc1() {
  lock(x) {
    lock(y) {
      ...
    }
  }
}
```

```
void proc2() {
  lock(y) {
    lock(x) {
      ...
    }
  }
}
```

Other Solutions

- There are a LOT:
  - Atomic operations
  - Semaphores
  - Monitors
  - **Software transactional memory**

Software Transactional Memory

- Very different approach from locks
- Code that needs to be run in a single unit is put into an **atomic block**
- Everything in an atomic block is executed in a single **transaction**

Transactions

- Execute the code in the atomic block
- If it did not conflict with anything, then **commit** it
- If there was a conflict, then **roll back** and **retry**
- All or nothing

Example With STM

```
int x = -1;
```

```
void proc1() {
  atomic {
    if (x == -1) {
      x = 5;
    }
  }
}
```

```
void proc2() {
  atomic {
    if (x == -1) {
      x = 6;
    }
  }
}
```

Not a Lock

- We do not explicitly state what we are locking on
- We only roll back if there was a change
  - With locks, we could lock something and never change it
  - Atomic blocks automatically determine what needs to be "locked"

Performance

- Scale **much** better than locks
  - Oftentimes conflicts are possible but infrequent, and performance hits are mostly at conflicts
  - Depending on the implementation, atomic blocks can have a much lower overhead than locking