Grading

8 Homeworks (40%)

5 programming assignments (20%)

3 Midterms (40%)
Collaboration

Permitted
- talking about homework problems with other students;
- using other textbooks;
- using the Internet.

NOT Permitted
- obtaining the answer directly from anyone or anything else in any form.
Moore’s Law Literally

- Transistor count still rising
- Clock speed flattening sharply
Moore’s Law Actually
Extinct: the Uniprocessor
Extinct: Shared Memory Multiprocessor (SMP)
Meet the New Boss: The Multicore Processor (CMP)

All on the same chip
Why is Kunle Smiling?

His new multicore
Why do we care?

Time no longer cures software bloat

The “free ride” is over

Double your program’s path length?

You can’t just wait 6 months

You must be twice as parallel!
Traditional Scaling Process

User code

Time: Moore's law
Ideal Scaling Process

Speedup

1.8x  3.6x  7x

User code

print ln("hello world!")

print ln("hello world!")

print ln

print ln

print ln

print ln

print ln

print ln

print ln

print ln

print ln

print ln

print ln

Multicore

Unfortunately, not so simple...
Actual Scaling Process

Speedup

User code

Multicore

Unfortunately, not so simple…
Multicore Programming: Course Overview

- Fundamentals
- Models, algorithms, impossibility
- Real-World programming
- Architectures
- Techniques
Sequential Computation

thread

memory
Concurrent Computation
Asynchrony

- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)
Model Summary

Multiple *threads*

Sometimes called *processes*

Single shared *memory*

*Objects* live in memory

Unpredictable asynchronous delays
Road Map

We will focus on *principles* first, then *practice*.

Start with *idealized* models.

Look at *simplistic* problems.

First *correctness*, then *performance*.

“Correctness may seem theoretical, but incorrectness has practical impact.”
Concurrency Jargon

**Hardware:** *processors, cores*

**Software:** *threads or processes*

Sometimes OK to conflate them, sometimes not
Parallel Primality Testing

Challenge

Print primes from 1 to $10^{10}$

10 cores

One thread per core

Goal: ten-fold speedup (or close)
Load Balancing

Split work evenly

Each thread tests range of $10^9$
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*109+1, j<(i+1)*109; j++) {
        if (isPrime(j))
            print(j);
    }
}
Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads uneven
- Hard to predict
Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads uneven
- Hard to predict
Shared Counter

each thread takes a number
Thread $i$

```java
int counter = new Counter(1);
void primePrint {
    long j = 0;
    while (j < 1010) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```
Procedure for Thread $i$

Counter counter = new Counter(1);
void primePrint {
    long j = 0;
    while (j < 1010) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
Where Things Live

```c
void primePrint() {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10^9; j < (i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
```

Local variables

shared counter

shared memory

code

cache

cache

cache

Bus
Thread \(i\)

Counter counter = new Counter(1);
void primePrint {
    long j = 0;
    \textbf{while} (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}

Stop when every value taken
Counter counter = new Counter(1);
void primePrint {
    long j = 0;
    while (j < 1010) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
    }
}
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
    }
}
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
    }
}
public class Counter {
    private long value;
    public long getAndIncrement() {
        return value++;
    }
}

temp = value;
value = temp + 1;
return temp;
Uh, oh

Value... 1 2 3 2

read 1
write 2
read 2
write 3
read 1
write 2

time
Is this problem inherent?

If we could only glue reads and writes together…
Challenge

public class Counter {
    private long value;
    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}

Make these steps **atomic** (indivisible)
public class Counter {
    private long value;
    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}

HW ReadModifyWrite() Instruction?
public class Counter {
    private long value;
    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
public class Counter {
    private long value;
    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
Mutual Exclusion, or “Alice & Bob share a pond”
Alice has a pet

Art of Multiprocessor Programming
Bob has a pet
The Problem

The pets don’t get along
Two kinds of Properties

- Safety
  - Nothing bad ever happens

- Liveness
  - Something good eventually happens
A Safety Property

Mutual Exclusion

Both pets never in pond simultaneously
A Liveness Property

No Deadlock

if only one wants in, it gets in

if both want in, one gets in
Simple Protocol

Idea

Just look at the pond

Gotcha!

Not atomic

(Trees obscure the view)
Interpretation

Threads can’t “see” what other threads are doing

Explicit communication required for coordination
Cell Phone Protocol

Idea

Bob calls Alice (or vice-versa)

Gotcha!

Bob takes shower

Alice recharges battery

Bob out shopping for pet food …
Interpretation

Message-passing doesn’t work

Recipient might not be there at all

Listening

Communication must be persistent (like writing)

Not transient (like speaking)
Can Protocol
Bob conveys a bit
Bob conveys a bit
Can Protocol

Idea

- Cans on Alice’s windowsill
- Strings lead to Bob’s house
- Bob pulls strings, knocks over cans

Gotcha

- Cans cannot be reused
- Bob runs out of cans
Interpretation

Cannot solve mutual exclusion with interrupts

Sender sets fixed bit in receiver’s space

Receiver resets bit when ready

Requires unbounded number of interrupt bits
Flag Protocol

Art of Multiprocessor Programming
Alice’s Protocol (sort of)
Bob’s Protocol (sort of)
Alice’s Protocol

- Raise flag
- Wait until Bob’s flag is down
- Unleash pet
- Lower flag when pet returns
Bob’s Protocol

- Raise flag
- Wait until Alice’s flag is down
- Unleash pet
- Lower flag when pet returns

*danger!*
Bob’s Protocol (2nd try)

Raise flag

While Alice’s flag is up

Lower flag

Wait for Alice’s flag to go down

Raise flag

Unleash pet

Lower flag when pet returns
Bob’s Protocol

Raise flag

While Alice’s flag is up

Lower flag

Wait for Alice’s flag to go down

Raise flag

Unleash pet

Lower flag when pet returns

Bob defers to Alice
The Flag Principle

Raise flag

Look at other’s flag

Flag Principle:

If each raises and looks, then

Last to look must see both flags up
Proof of Mutual Exclusion

Assume both pets in pond

Derive a contradiction …

By reasoning *backwards*
Proof of Mutual Exclusion

Consider the last time Alice and Bob each looked before letting the pets in

Without loss of generality assume Alice was the last to look…
Proof

Alice last raised her flag

Bob last raised flag

Bob’s last look

Alice’s last look

Alice must have seen Bob’s Flag. A Contradiction
Proof of No Deadlock

If only one pet wants in, it gets in.

Deadlock means both continually trying to get in.

If Bob sees Alice’s flag, he backs off, gives her priority (Alice’s lexicographic privilege)

QED
Remarks

Protocol is *unfair*

Bob’s pet might never get in

Protocol uses *waiting*

If Bob is eaten by his pet,

Alice’s pet might never get in
Moral of Story

Mutual Exclusion *cannot* be solved by

- transient communication (cell phones)
- interrupts (cans)

It *can* be solved by

- one-bit shared variables

that can be read or written
The Arbiter Problem (an aside)

Pick a point

Pick a point
The Fable Continues

Alice and Bob fall in love & marry

Then they fall out of love & divorce

She gets the pets

He has to feed them

Leading to a new coordination problem:

Producer-Consumer
Bob Puts Food in the Pond
Alice releases her pets to Feed
Producer/Consumer

Alice and Bob can’t meet

Each has restraining order on other

So he puts food in the pond

And later, she releases the pets

Must avoid

Releasing pets when there’s no food

Putting out food if uneaten food remains
Producer/Consumer

Need a mechanism that ensures

Bob lets Alice know when food has been put out

Alice lets Bob know when to put out more food
Surprise Solution
Bob puts food in Pond
Bob knocks over Can

Art of Multiprocessor Programming
Alice Releases Pets
Alice Resets Can when Pets are Fed
Pseudocode

while (true) {
    while (can.isUp()){
    pet.release();
    pet.recapture();
    can.reset();
}
}
Pseudocode

Alice’s code

```java
while (true) {
    while (can.isUp()) {
        pet.release();
        pet.recapture();
        can.reset();
    }
}
```

Bob’s code

```java
while (true) {
    while (can.isDown()) {
        pond.stockWithFood();
        can.knockOver();
    }
}
```
**Correctness**

**Mutual Exclusion**
Pets and Bob never together in pond

**No Starvation**
If Bob always willing to feed, and pets always famished, then pets eat infinitely often.

**Producer/Consumer**
The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.
Could Also Solve Using Flags
Both solutions use *waiting*

\[\text{while (condition) \{doNothing()\}}\]

Sometimes waiting is *problematic*

If one party is delayed …

So is everyone else

But delays are common & unpredictable
The Fable drags on ... 

Bob and Alice still have issues

So they need to communicate

They agree to use billboards ...
Billboards are Large

Letter Tiles
From Scrabble™ box

Art of Multiprocessor Programming
Write One Letter at a Time ...
To post a message

W A S H T H E C A R

whew
Let’s send another message
Uh-Oh

SELL THE CAR

OK

Art of Multiprocessor Programming
Readers/Writers

Devise a protocol so that

- Writer writes one letter at a time
- Reader reads one letter at a time
- Reader sees “snapshot”
- Old message or new message
- No mixed messages
Readers/Writers (continued)

Easy with mutual exclusion

But mutual exclusion requires *waiting*

One waits for the other

Everyone executes sequentially

Remarkably

We can solve R/W without mutual exclusion
Amdahl’s Law

Speedup = \frac{1\text{-thread execution time}}{n\text{-thread execution time}}
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

Speedup = \frac{1}{1 - p + \frac{p}{n}}

Sequential fraction

Parallel fraction

Number of threads
Amdahl’s Law

Bad synchronization ruins everything
Example

Ten cores

60% concurrent, 40% sequential

How close to 10-fold speedup?

Speedup = 2.17
Example

Ten cores

80% concurrent, 20% sequential

How close to 10-fold speedup?

Speedup = 3.57
Example

Ten cores

90% concurrent, 10% sequential

How close to 10-fold speedup?

Speedup = 5.26
Example

Ten cores

99% concurrent, 1% sequential

How close to 10-fold speedup?

Speedup = 9.17
Actual Scaling Process

What happens if you don’t reduce sequential % of code
Shared Data Structures

Honk! Honk! Honk!

Why only 2.9 speedup

Coarse Grained

Fine Grained

25% Shared

75% Unshared

25% Shared

75% Unshared

25% Shared

75% Unshared
Shared Data Structures

- Coarse Grained
  - 25% Shared
  - 75% Unshared

- Fine Grained
  - 25% Shared
  - 75% Unshared

Why fine-grained parallelism matters
This course is about the parts that are hard to make concurrent ... but still have a big influence on speedup!
one

two

three

four

five

Art of Multiprocessor Programming