Principles of Software Construction: Objects, Design, and Concurrency
Part 6: Concurrency

The Perils of Concurrency

Can't live with it…

Can't live without it…

Christian Kästner **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

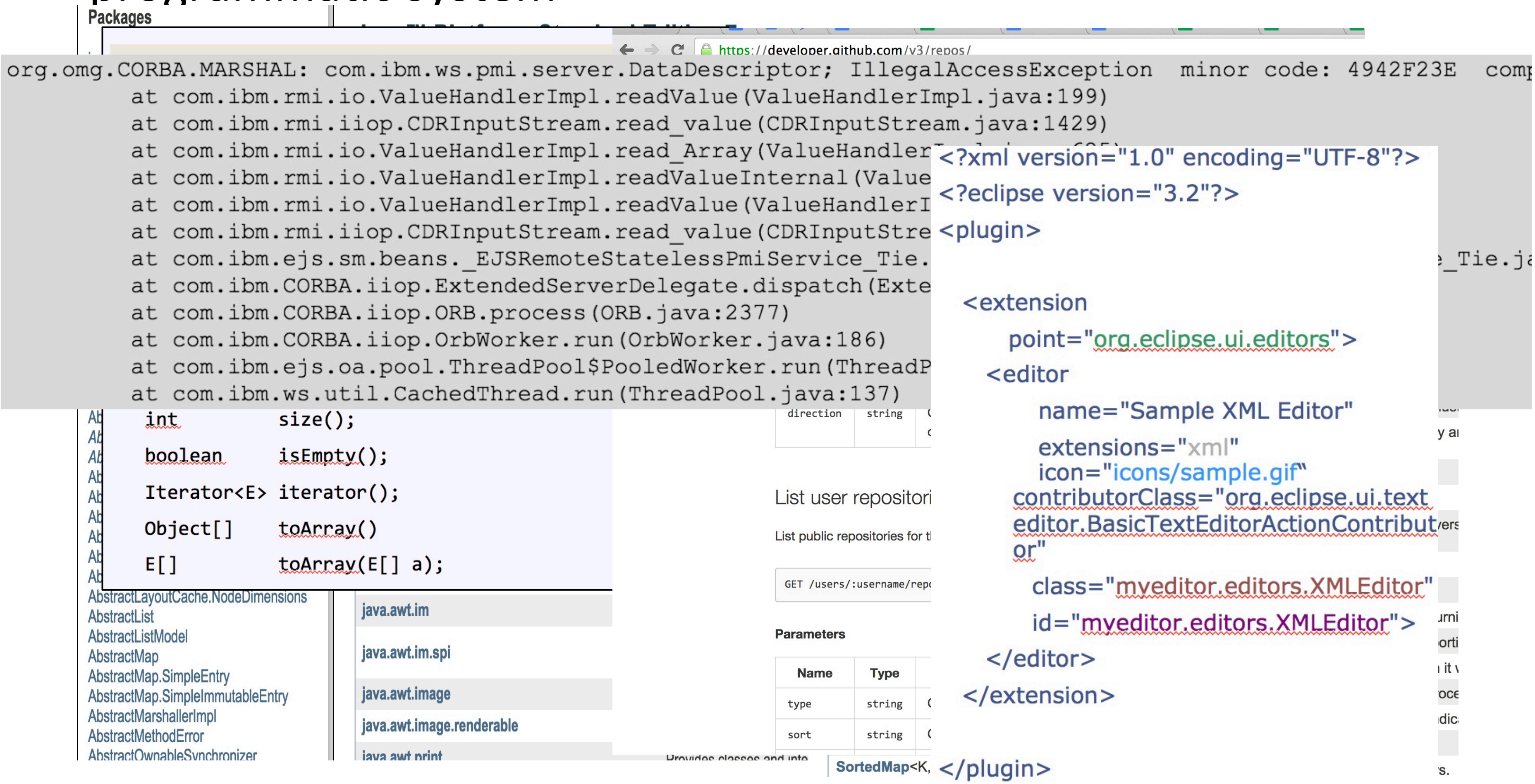
Administrivia

- Homework 5 team signups due tonight
- 2nd midterm exam Thursday
  - Review session tonight 7 – 9 p.m. in Hamburg Hall 1000
- Homework 5 framework design advice… (at the end of class)

Key concepts from last Thursday

API: Application Programming Interface

- An API defines the boundary between components/modules in a programmatic system



An API design process

- Define the scope of the API
  - Collect use-case stories, define requirements
  - Be skeptical
    - Distinguish true requirements from so-called solutions
    - "When in doubt, leave it out."
- Draft a specification, gather feedback, revise, and repeat
  - Keep it simple, short
- Code early, code often
  - Write *client code* before you implement the API

Key design principle: Information hiding

- "When in doubt, leave it out."

Minimize mutability

- Immutable objects are:
  - Inherently thread-safe
  - Freely shared without concern for side effects
  - Convenient building blocks for other objects
  - Can share internal implementation among instances
    - See `java.lang.String`
- Mutable objects require careful management of visibility and side effects
  - e.g. `Component.getSize()` returns a mutable `Dimension`
- Document mutability
  - Carefully describe state space

Course themes

- Code-level design
  - Process – how to start
  - Patterns – re-use conceptual solutions
  - Criteria – e.g. evolveability, understandability
- Analysis and modeling
  - Practical specification techniques and verification tools
- Object-oriented programming
  - Evolveability, reuse
  - Industry use – basis for frameworks
  - Vehicle is Java –industry, upper-division courses
- Threads and Concurrency
  - System abstraction – background computing
  - Performance
  - Our focus: explicit, application-level concurrency
    - Cf. functional parallelism (150, 210) and systems concurrency (213)

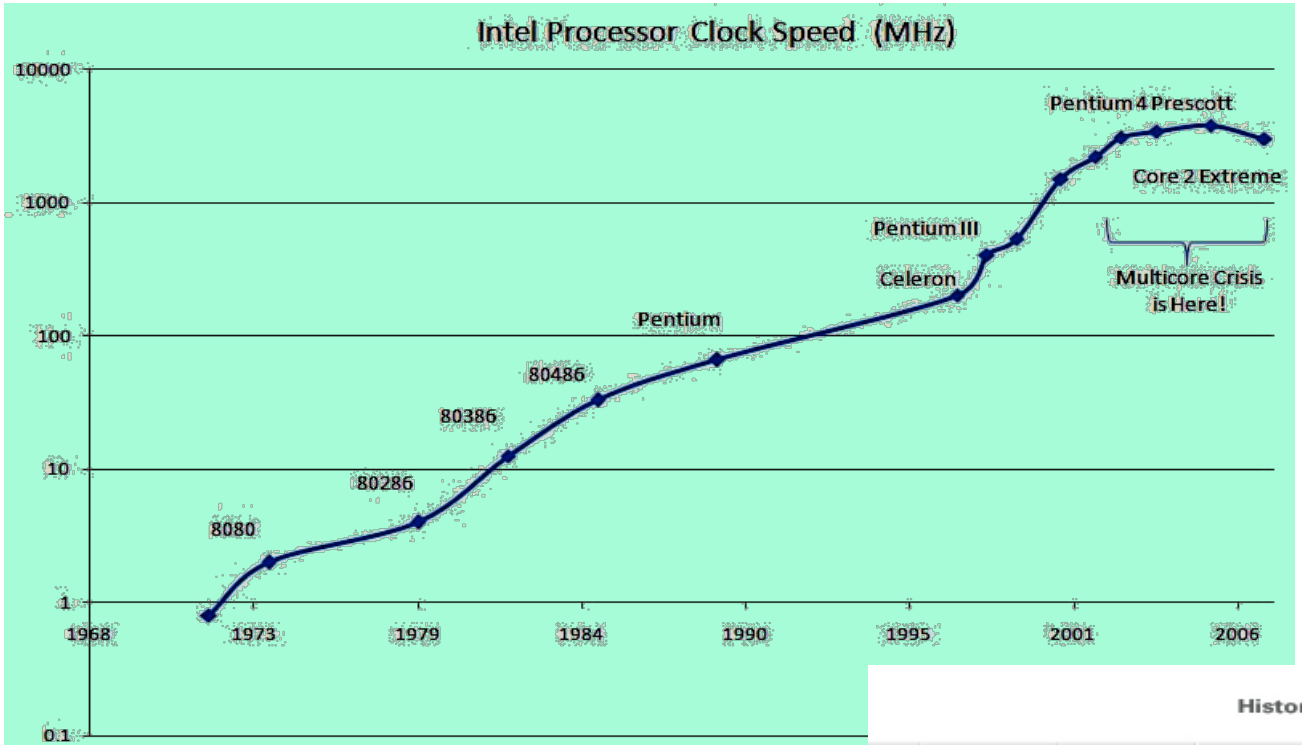
Today: Concurrency, part 1

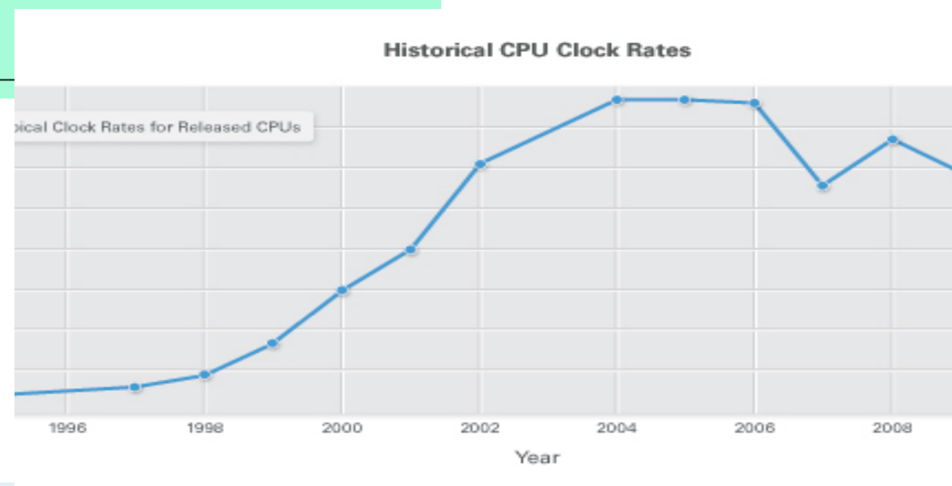
- The backstory
  - Motivation, goals, problems, …
- Basic concurrency in Java
  - Synchronization
- Coming soon (but not today):
  - Higher-level abstractions for concurrency
    - Data structures
    - Computational frameworks

Learning goals

- Understand concurrency as a source of complexity in software
- Know common abstractions for parallelism and concurrency, and the trade-offs among them
  - Explicit concurrency
    - Write thread-safe concurrent programs in Java
    - Recognize data race conditions
  - Know common thread-safe data structures, including high-level details of their implementation
  - Understand trade-offs between mutable and immutable data structures
  - Know common uses of concurrency in software design

Processor speeds over time



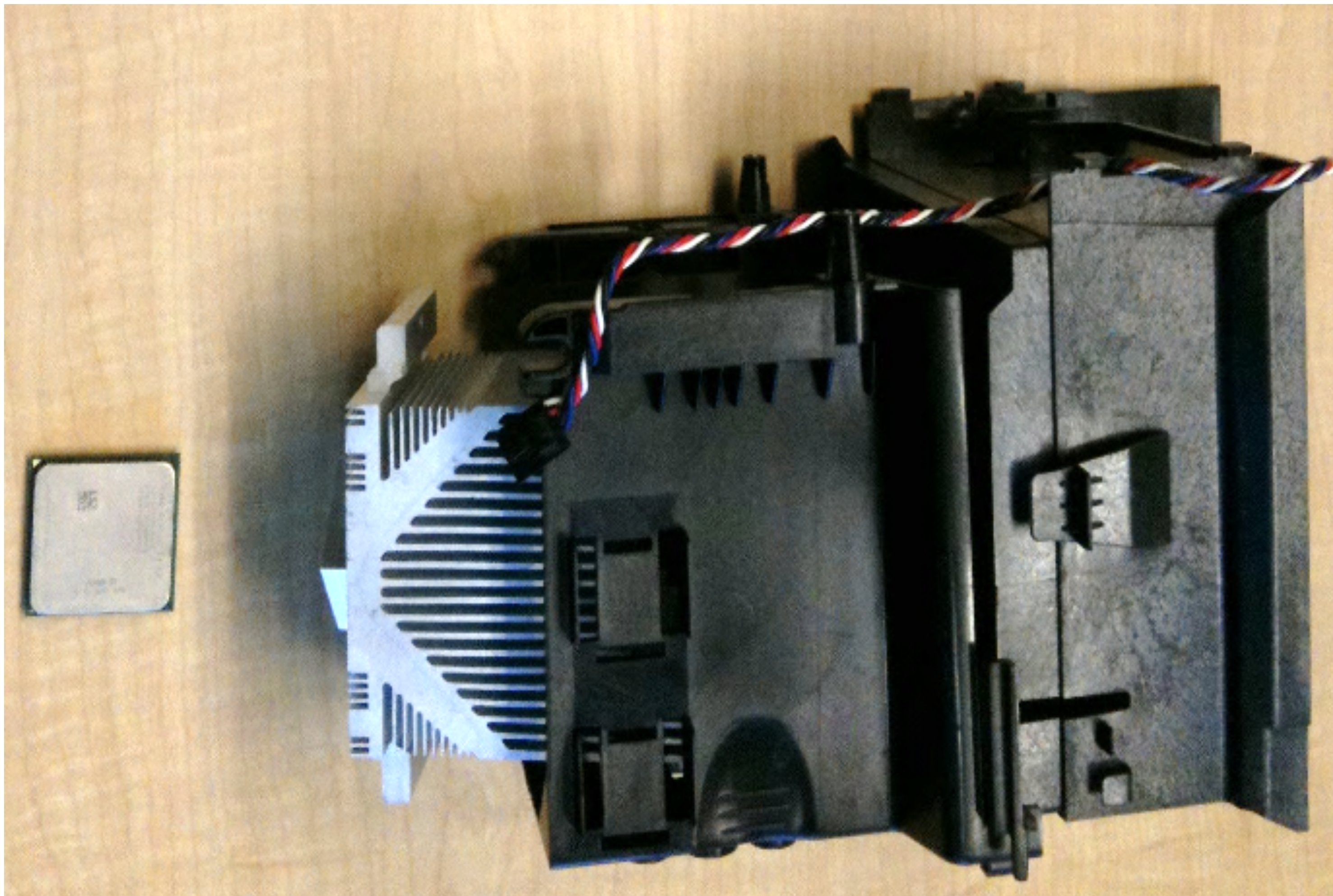


Power requirements of a CPU

- Approx.: **C**apacitance * **V**oltage^2 * **F**requency
- To increase performance:
  - More transistors, thinner wires: more **C**
    - More power leakage: increase **V**
  - Increase clock frequency **F**
    - Change electrical state faster: increase **V**
- Problem: Power requirements are super-linear to performance
  - Heat output is proportional to power input

One option: fix the symptom

- Dissipate the heat



One option: fix the symptom

- Better: Dissipate the heat with liquid nitrogen
  - Overclocking by Tom's Hardware's 5 GHz project

http://www.tomshardware.com/reviews/5-ghz-project,731-8.html

Another option: fix the underlying problem

- Reduce heat by limiting power input
  - Adding processors increases power requirements linearly with performance
    - Reduce power requirement by reducing the frequency and voltage
    - Problem: requires concurrent processing

Aside: Three sources of disruptive innovation

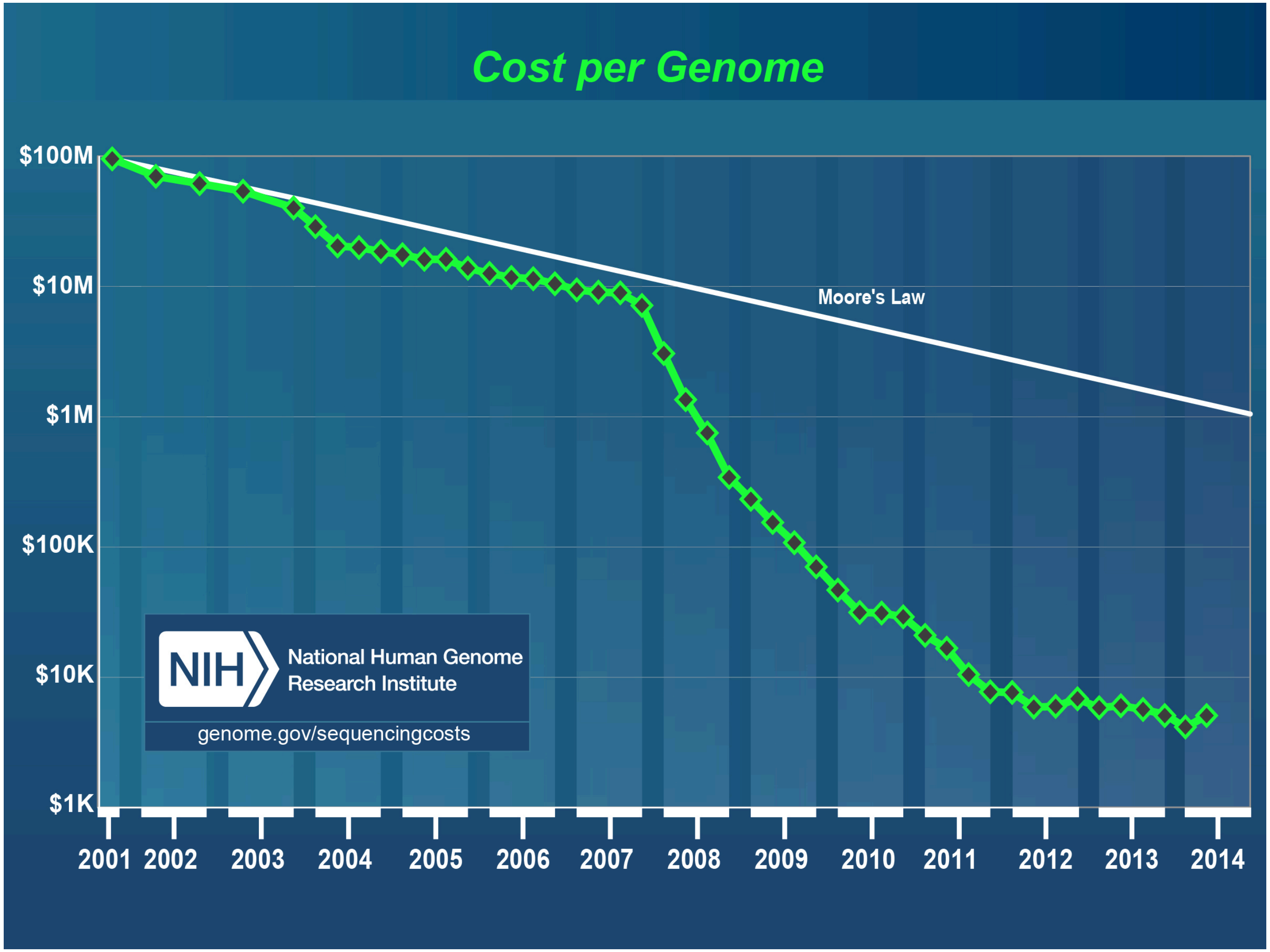
- Growth crosses some threshold
  - e.g., Concurrency: ability to add transistors exceeded ability to dissipate heat
- Colliding growth curves
  - Rapid design change forced by jump from one curve onto another
- Network effects
  - Amplification of small triggers leads to rapid change

Aside: The threshold for distributed computing

- Too big for a single computer?
  - Forces use of distributed architecture
    - Shifts responsibility for reliability from hardware to software
      - Allows you to buy larger cluster of cheap flaky machines instead of expensive slightly-less-flaky machines
        - » Revolutionizes data center design

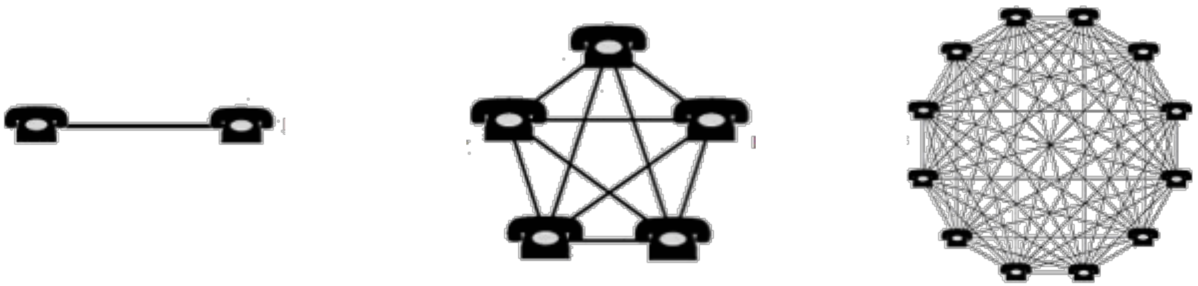
Aside: Colliding growth curves

- From http://www.genome.gov/sequencingcosts/



Aside: Network effects

- Metcalfe's rule: network value grows quadratically in the number of nodes
  - a.k.a. Why my mom has a Facebook account
  - `n(n-1)/2` potential connections for `n` nodes



  - Creates a strong imperative to merge networks
    - Communication standards, media formats, ...

Concurrency

- Simply: doing more than one thing at a time
  - In software: more than one point of control
    - Threads, processes
- Resources simultaneously accessed by more than one thread or process

Concurrency then and now

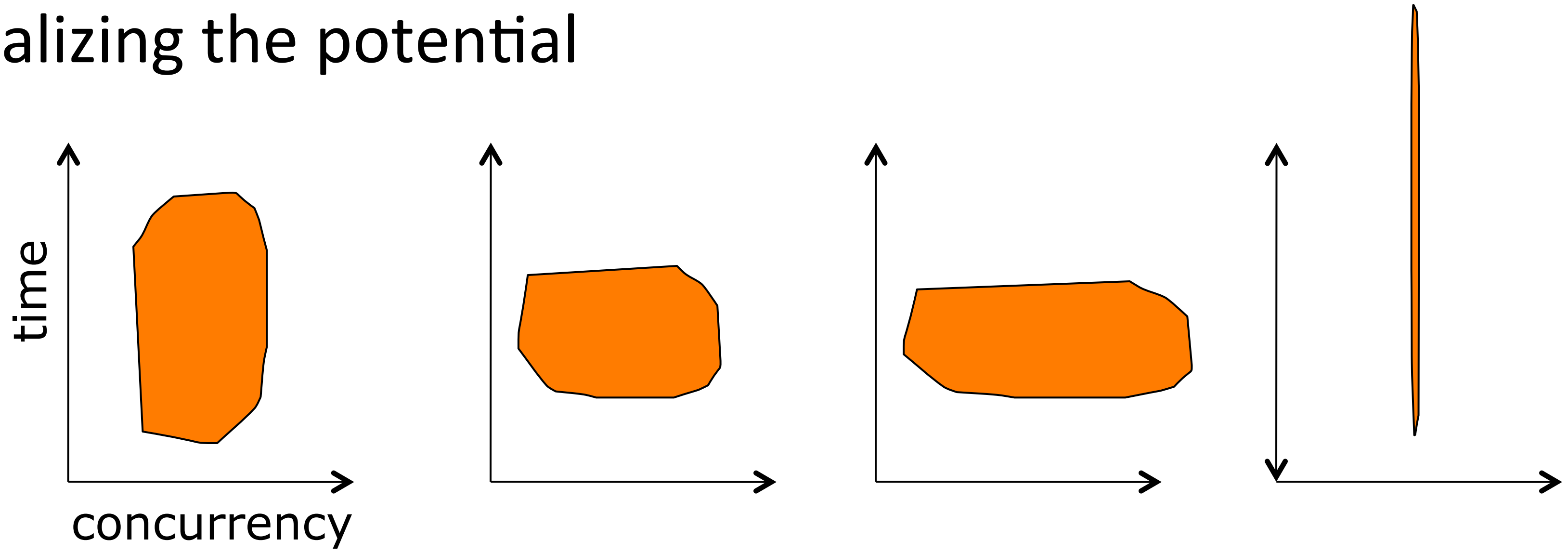
- In the past multi-threading was just a convenient abstraction
  - GUI design: event threads
  - Server design: isolate each client's work
  - Workflow design: producers and consumers
- Now: must use concurrency for scalability and performance

Image Name	Threads
IPSSVC.EXE	86
svchost.exe	82
System	80
afsd_service.exe	51
Rtvscan.exe	47
winlogon.exe	39
explorer.exe	20
ccEvtMgr.exe	19
svchost.exe	18
lsass.exe	18
tabtip.exe	17
svchost.exe	17
firefox.exe	16
services.exe	16
thunderbird.exe	15
csrss.exe	13
tcserver.exe	10
KeyboardSurroga...	10
spoolsv.exe	10
tvt_reg_monitor_...	10
svchost.exe	10
POWERPNT.EXE	9
taskmgr.exe	8
VPTray.exe	8
S24EvMon.exe	8
EvtEng.exe	8
emacs.exe	7
tvtsched.exe	7
ibmpmsvc.exe	7
AcroRd32.exe	7
vpngui.exe	6
cvpnd.exe	6
AluSchedulerSvc....	6
ccSetMgr.exe	6
svchost.exe	6
wisptis.exe	5
alg.exe	5
TPHKMGR.exe	5
ASRSVC.exe	5

Problems of concurrency

- Realizing the potential
  - Keeping all threads busy doing useful work
- Delivering the right language abstractions
  - How do programmers think about concurrency?
  - Aside: parallelism vs. concurrency
- Non-determinism
  - Repeating the same input can yield different results

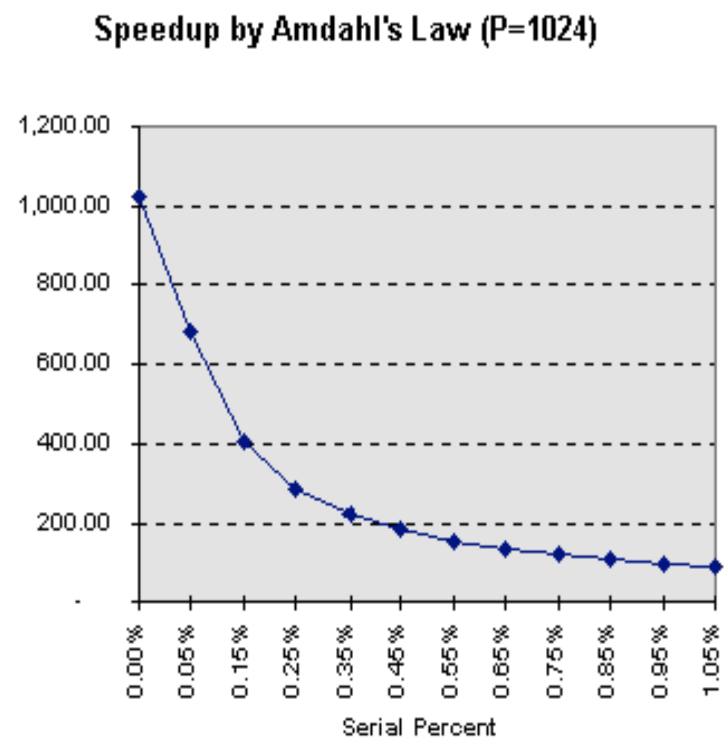
Realizing the potential



- Possible metrics of success
  - Breadth: extent of simultaneous activity
    - width of the shape
  - Depth (or span): length of longest computation
    - height of the shape
  - Work: total effort required
    - area of the shape
- What are the typical goals in parallel algorithm design?

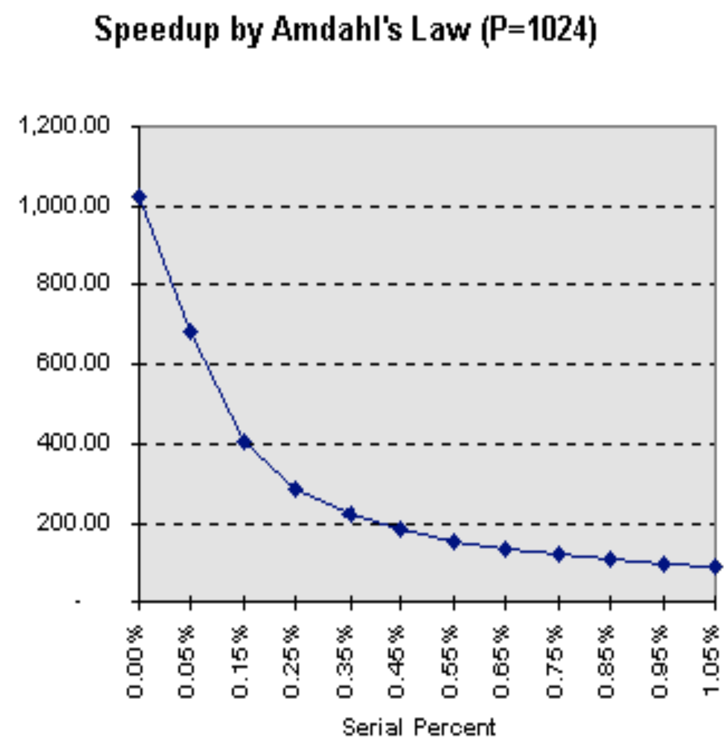
Amdahl's law: How good can the depth get?

- Ideal parallelism with `N` processors:
  - Speedup = `N`
- In reality, some work is always inherently sequential
  - Let `F` be the portion of the total task time that is inherently sequential
  - Speedup = $$\frac{1}{F + (1 - F)/N}$$
  - Suppose F = 10%. What is the max speedup? (you choose `N`)



Amdahl's law: How good can the depth get?

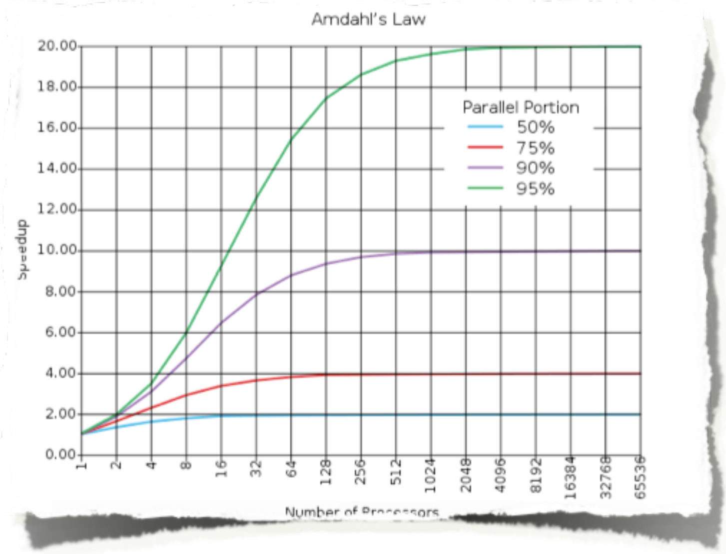
- Ideal parallelism with N processors:
  - Speedup = N
- In reality, some work is always inherently sequential
  - Let F be the portion of the total task time that is inherently sequential
  - Speedup = $\frac{1}{F + (1 - F)/N}$



  - Suppose F = 10%. What is the max speedup? (you choose N)
    - As N approaches ∞, 1/(0.1 + 0.9/N) approaches 10.

institute for SOFTWARE RESEARCH

Using Amdahl's law as a design guide



- For a given algorithm, suppose
  - `N` processors
  - Problem size `M`
  - Sequential portion `F`
- An obvious question:
  - What happens to speedup as `N` scales?
- A less obvious, important question:
  - What happens to `F` as problem size `M` scales?

"For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law."

— Doron Rajwan, Intel Corp

Principles of Software Construction: Objects, Design, and Concurrency
Part 6: Concurrency, Part 2

The Perils of Concurrency

Can't live with it…

Can't live without it…

Christian Kästner **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 5a due tomorrow 9 a.m.
- 2nd midterm exam returned today at end of class
- Do you want to be a software engineer?

The foundations of the Software Engineering minor

- Core computer science fundamentals
- Building good software
- Organizing a software project
  - Development teams, customers, and users
  - Process, requirements, estimation, management, and methods
- The larger context of software
  - Business, society, policy
- Engineering experience
- Communication skills
  - Written and oral

SE minor requirements

- Prerequisite: 15-214
- Two core courses
  - 15-313 Foundations of SE (fall semesters)
  - 15-413 SE Practicum (spring semesters)
- Three electives
  - Technical
  - Engineering
  - Business or policy
- Software engineering internship + reflection
  - 8+ weeks in an industrial setting, then
  - 17-413

To apply to be a Software Engineering minor

- Email aldrich@cs.cmu.edu and clegoues@cs.cmu.edu
  - Your name, Andrew ID, class year, QPA, and minor/majors
  - Why you want to be a SE minor
  - Proposed schedule of coursework

- Spring applications due by Friday, 10 Apr 2015
  - Only 15 SE minors accepted per graduating class
- More information at:
  - http://isri.cmu.edu/education/undergrad/

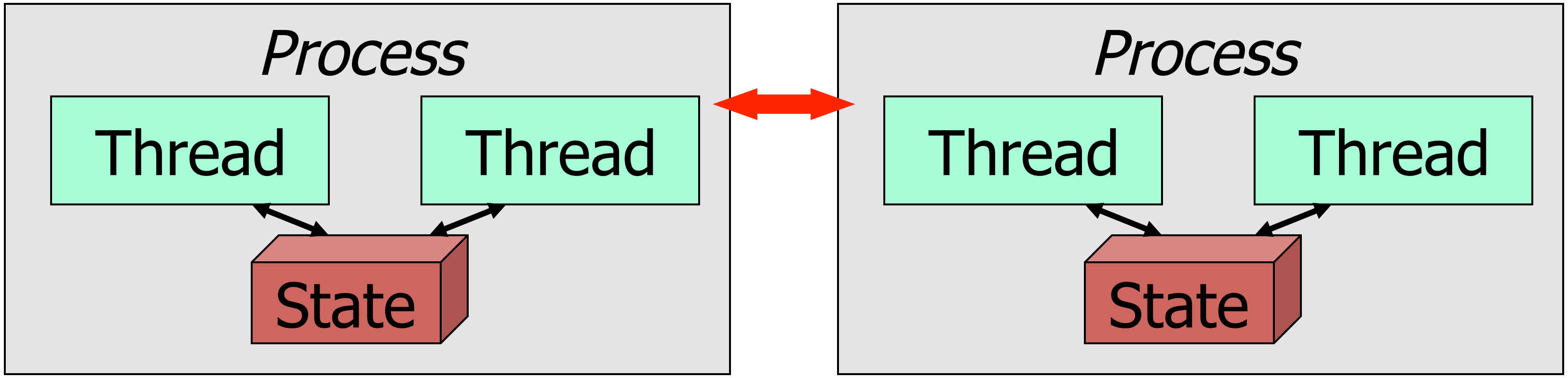
Key concepts from last Tuesday

Today: Concurrency, part 2

- The backstory
  - Motivation, goals, problems, …
- Basic concurrency in Java
  - Synchronization
- Coming soon:
  - Higher-level abstractions for concurrency
    - Data structures
    - Computational frameworks

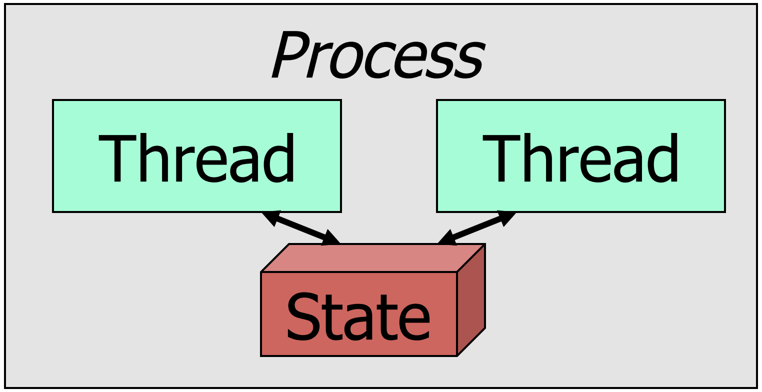
Abstractions of concurrency

- Processes
  - Execution environment is isolated
    - Processor, in-memory state, files, …
  - Inter-process communication typically slow, via message passing
    - Sockets, pipes, …
- Threads
  - Execution environment is shared
  - Inter-thread communication typically fast, via shared state

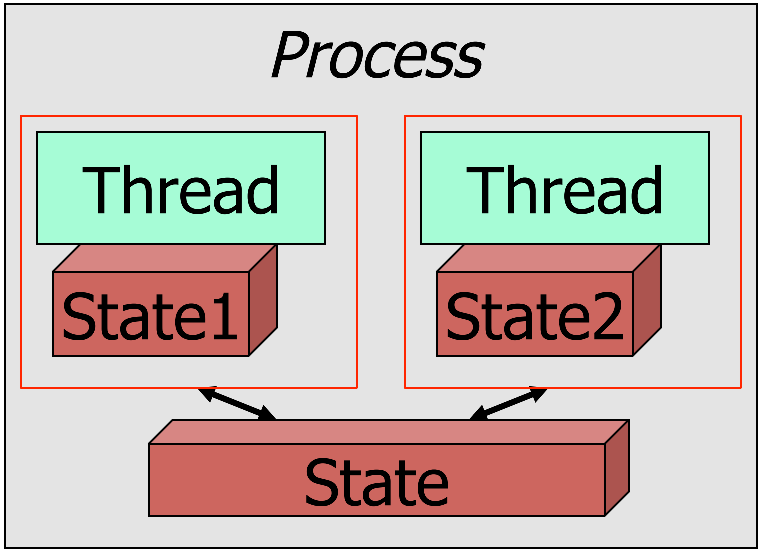


Aside: Abstractions of concurrency

- What you see:
  - State is all shared



- A (slightly) more accurate view of the hardware:
  - Separate state stored in registers and caches
  - Shared state stored in caches and memory



Basic concurrency in Java

- The `java.lang.Runnable` interface

```
void           run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void           start();
static void    sleep(long millis);
void           join();
boolean        isAlive();
static Thread  currentThread();
```

- See IncrementTest.java

Atomicity

- An action is *atomic* if it is indivisible
  - Effectively, it happens all at once
    - No effects of the action are visible until it is complete
    - No other actions have an effect during the action
- In Java, integer increment is not atomic

```
i++;
```

is actually

1. Load data from variable `i`
2. Increment data by `1`
3. Store data to variable `i`

One concurrency problem: race conditions

- A *race condition* is when multiple threads access shared data and unexpected results occur depending on the order of their actions
- E.g., from IncrementTest.java:
  - Suppose `classData` starts with the value 41:

Thread A:

```
classData++;
```

Thread B:

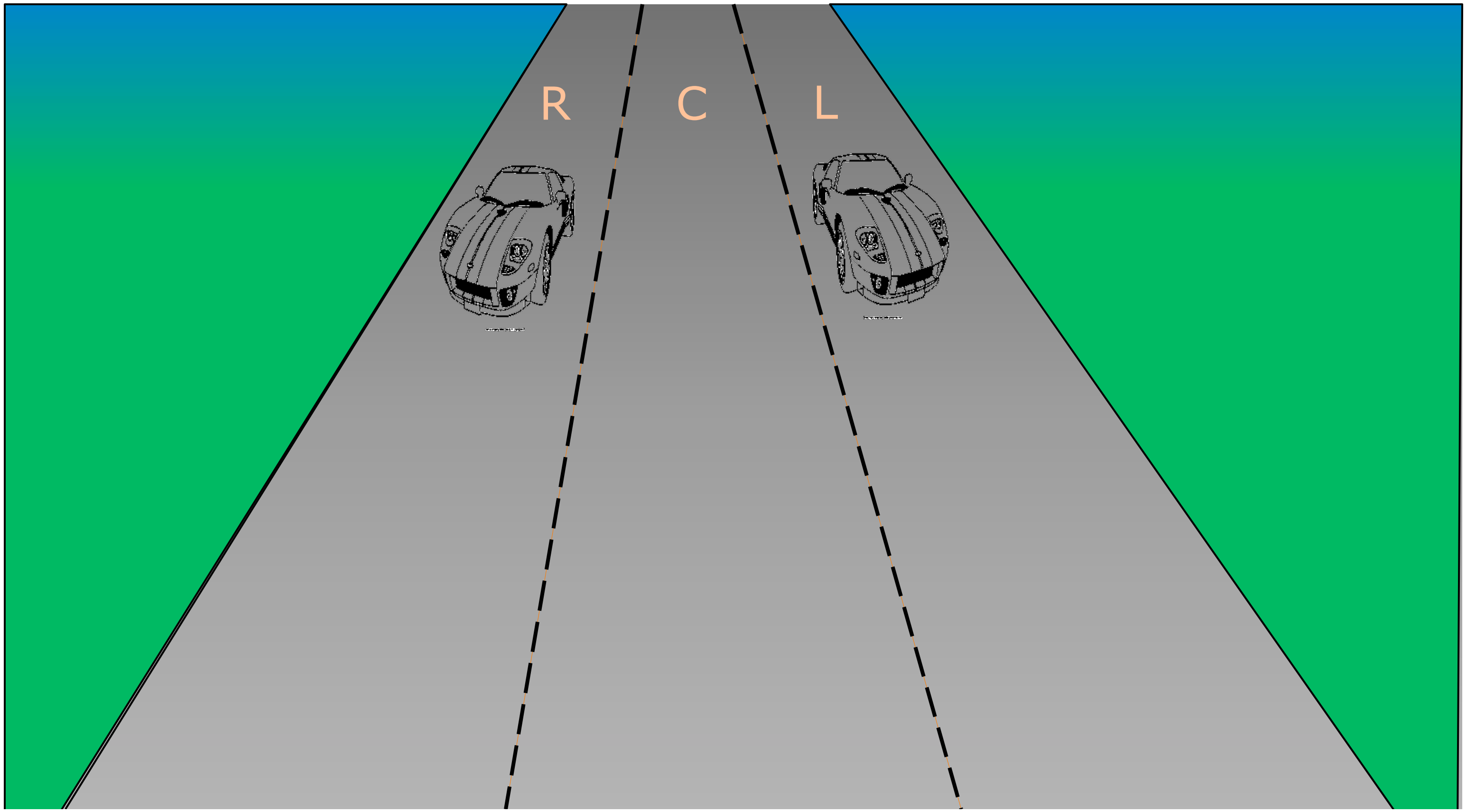
```
classData++;
```

One possible interleaving of actions:

1A. Load data(41) from `classData`
1B. Load data(41) from `classData`
2A. Increment data(41) by 1 -> 42
2B. Increment data(41) by 1 -> 42
3A. Store data(42) to `classData`
3B. Store data(42) to `classData`

Race conditions in real life

- E.g., check-then-act on the highway



Race conditions in real life

- E.g., check-then-act at the bank
  – The "debit-credit problem"

Alice, Bob, Bill, and the *Bank*

- A. *Alice* to pay *Bob* $30
  - Bank actions
    1. Does Alice have $30 ?
    2. Give $30 to *Bob*
    3. Take $30 from *Alice*

- B. *Alice* to pay *Bill* $30
  - Bank actions
    1. Does Alice have $30 ?
    2. Give $30 to *Bill*
    3. Take $30 from *Alice*

- If *Alice* starts with $40, can *Bob* and *Bill* both get $30?

Race conditions in real life

- E.g., check-then-act at the bank
  - The "debit-credit problem"

Alice, Bob, Bill,* and the *Bank

- **A. *Alice* to pay *Bob* $30**
  - **Bank actions**
    1. **Does Alice have $30 ?**
    2. **Give $30 to *Bob***
    3. **Take $30 from *Alice***
- **B. *Alice* to pay *Bill* $30**
  - **Bank actions**
    1. **Does Alice have $30 ?**
    2. **Give $30 to *Bill***
    3. **Take $30 from *Alice***
- **If *Alice* starts with $40, can *Bob* and *Bill* both get $30?**

A.1
A.2
B.1
B.2
A.3
B.3!

Race conditions in *your* life

- E.g., check-then-act in simple code

```
public class StringConverter {
    private Object o;
    public void set(Object o) {
        this.o = o;
    }
    public String get() {
        if (o == null) return "null";
        return o.toString();
    }
}
```

  – See StringConverter.java, Getter.java, Setter.java

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

i: **00000…00000111**

⋮

i: **00000…00101010**

Some actions are atomic

Precondition:

```
int i = 7;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

- What are the possible values for `ans`?

i: **00000…00000111**

⋮

i: **00000…00101010**

- In Java:
  - Reading an int variable is atomic
  - Writing an int variable is atomic
  - Thankfully, ans: **00000…00101111** is not possible

Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

high bits	**low bits**

  - Concurrently:
    - Thread A writing high bits and low bits
    - Thread B reading high bits and low bits

Precondition:

```
long i = 10000000000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

`ans:` **01001…00000000** (10000000000)

`ans:` **00000…00101010** (42)

`ans:` **01001…00101010** (10000000042 or …)

Primitive concurrency control in Java

- Each Java object has an associated intrinsic lock
  - All locks are initially unowned
  - Each lock is *exclusive*: it can be owned by at most one thread at a time
- The `synchronized` keyword forces the current thread to obtain an object's intrinsic lock
  - E.g.,

```
synchronized void foo() { … } // locks "this"

synchronized(fromAcct) {
         if (fromAcct.getBalance() >= 30) {
        toAcct.deposit(30);
        fromAcct.withdrawal(30);
    }
 }
```

- See SynchronizedIncrementTest.java

Primitive concurrency control in Java

- `java.lang.Object` allows some coordination via the intrinsic lock:

```
void wait();
void wait(long timeout);
void wait(long timeout, int nanos);
void notify();
void notifyAll();
```

- See Blocker.java, Notifier.java, NotifyExample.java

Primitive concurrency control in Java

- Each lock can be owned by only one thread at a time
- Locks are *re-entrant*: If a thread owns a lock, it can lock the lock multiple times
- A thread can own multiple locks

```
synchronized(lock1) {
    // do stuff that requires lock1

    synchronized(lock2) {
        // do stuff that requires both locks
     }

     // …
 }
```

Another concurrency problem: deadlock

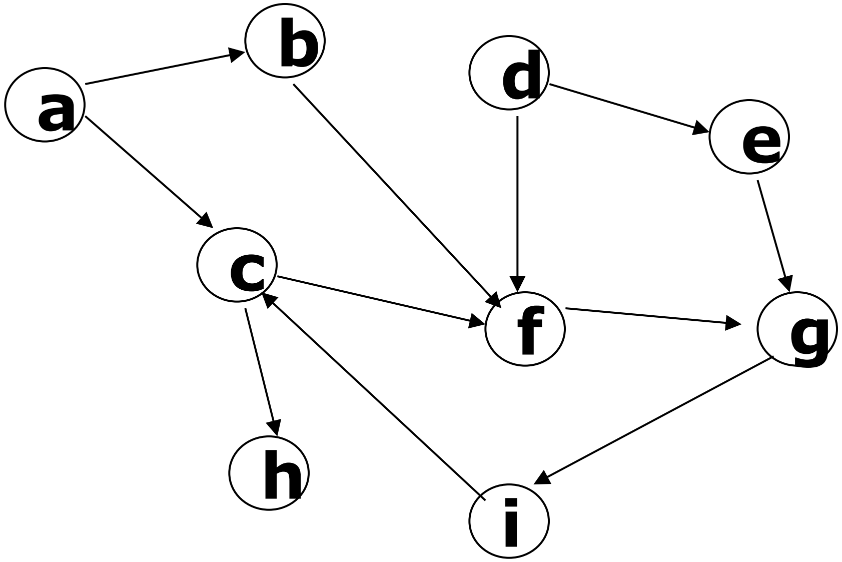
- E.g., Alice and Bob, unaware of each other, both need file *A* and network connection *B*
  - Alice gets lock for file *A*
  - Bob gets lock for network connection *B*
  - Alice tries to get lock for network connection *B*, and waits…
  - Bob tries to get lock for file *A*, and waits…
- See Counter.java and DeadlockExample.java

Dealing with deadlock (abstractly, not with Java)

- Detect deadlock
  - Statically?
  - Dynamically at run time?
- Avoid deadlock
- Alternative approaches
  - Automatic restarts
  - Optimistic concurrency control

Detecting deadlock with the waits-for graph

- The *waits-for graph* represents dependencies between threads
  - Each node in the graph represents a thread
  - A directed edge T1->T2 represents that thread T1 is waiting for a lock that T2 owns
- Deadlock has occurred iff the waits-for graph contains a cycle



Deadlock avoidance algorithms

- Prevent deadlock instead of detecting it
  - E.g., impose total order on all locks, require locks acquisition to satisfy that order
    - Thread:

```
acquire(lock1)
acquire(lock2)
acquire(lock9)
acquire(lock42)  // now can't acquire lock30, etc…
```

Avoiding deadlock with restarts

- One option: If thread needs a lock out of order, restart the thread
  - Get the new lock in order this time
- Another option: Arbitrarily kill and restart long-running threads

Avoiding deadlock with restarts

- One option: If thread needs a lock out of order, restart the thread
  - Get the new lock in order this time
- Another option: Arbitrarily kill and restart long-running threads
- Optimistic concurrency control
  - e.g., with a copy-on-write system
  - Don't lock, just detect conflicts later
    - Restart a thread if a conflict occurs

Another concurrency problem: livelock

- In systems involving restarts, *livelock* can occur
  - Lack of progress due to repeated restarts
- *Starvation*: when some task(s) is(are) repeatedly restarted because of other tasks

Principles of Software Construction: Objects, Design, and Concurrency

Part 6: Concurrency, Part 3

The Perils of Concurrency

Can't live with it…

Can't live without it…

Christian Kästner **Charlie Garrod**

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 5b due next Thursday, 11:59 p.m.
  - Finish by Friday (10 Apr) 10 a.m. if you want to be considered as a "Best Framework" for Homework 5c
    - Our evaluation considers:
      - Novelty
      - Functional correctness
      - Documentation
      - …

Key concepts from Tuesday

Bad news: some simple actions are not atomic

- Consider a single 64-bit `long` value

high bits	low bits

  - Concurrently:
    - Thread A writing high bits and low bits
    - Thread B reading high bits and low bits

Precondition:

```
long i = 10000000000;
```

Thread A:

```
i = 42;
```

Thread B:

```
ans = i;
```

`ans:` **01001...00000000** (10000000000)

`ans:` **00000...00101010** (42)

`ans:` **01001...00101010** (10000000042 or ...)

Key concepts from Tuesday

- Basic concurrency in Java
- Atomicity
- Race conditions
- The Java `synchronized` keyword

The Java *happens-before* relation

- Java guarantees a transitive, consistent order for some memory accesses
  - Within a thread, one action *happens-before* another action based on the usual program execution order
  - Release of a lock *happens-before* acquisition of the same lock
  - `Object.notify` *happens-before* `Object.wait` returns
  - `Thread.start` *happens-before* any action of the started thread
  - Write to a `volatile` field *happens-before* any subsequent read of the same field
  - …
- Assures ordering of reads and writes
  - A race condition can occur when reads and writes are not ordered by the happens-before relation

Concurrency control in Java

- Using primitive synchronization, you are responsible for correctness:
  - Avoiding race conditions
  - Progress (avoiding deadlock)
- Java provides tools to help:
  - `java.util.concurrent.atomic`
  - `java.util.concurrent`

The power of immutability

- Recall: Data is *mutable* if it can change over time. Otherwise it is *immutable*.
  - Primitive data declared as `final` is always immutable
- After immutable data is initialized, it is immune from race conditions

The `java.util.concurrent.atomic` package

- Concrete classes supporting atomic operations
  - `AtomicInteger`

```
int  get();
void set(int newValue);
int  getAndSet(int newValue);
int  getAndAdd(int delta);
boolean compareAndSet(int expectedValue,
                      int newValue);

…
```

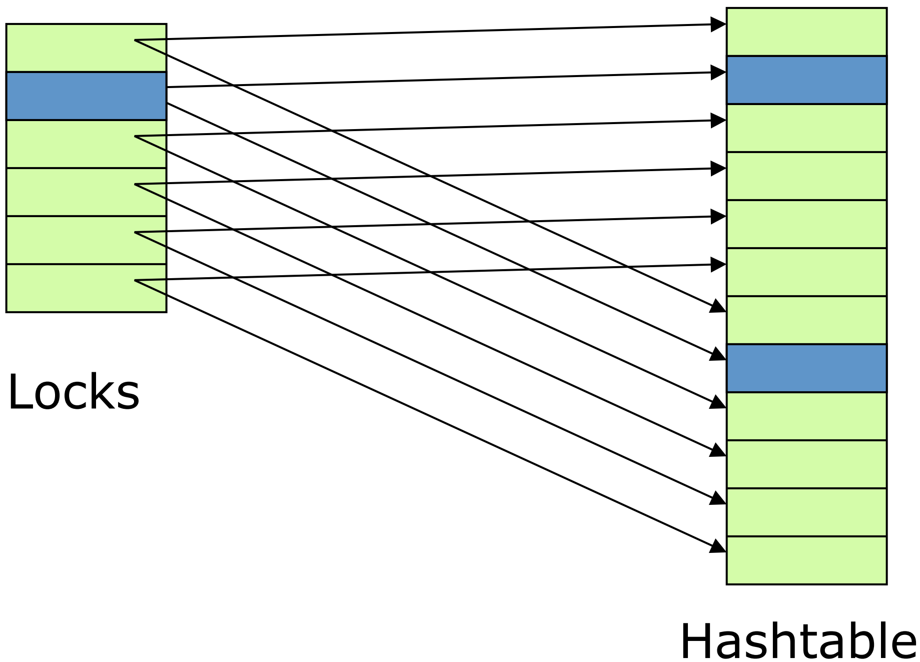
  - `AtomicIntegerArray`
  - `AtomicBoolean`
  - `AtomicLong`
  - …

The `java.util.concurrent` package

- Interfaces and concrete thread-safe data structure implementations
  - `ConcurrentHashMap`
  - `BlockingQueue`
    - `ArrayBlockingQueue`
    - `SynchronousQueue`
  - `CopyOnWriteArrayList`
  - …
- Other tools for high-performance multi-threading
  - `ThreadPools` and `Executor` services
  - `Locks` and `Latches`

java.util.concurrent.ConcurrentHashMap

- Implements `java.util.Map<K,V>`
  - High concurrency lock striping
    - Internally uses multiple locks, each dedicated to a region of the hash table
    - Locks just the part of the table you actually use
    - You use the `ConcurrentHashMap` like any other map…



java.util.concurrent.BlockingQueue

- Implements java.util.Queue<E>
- java.util.concurrent.SynchronousQueue
  - Each put directly waits for a corresponding poll
  - Internally uses wait/notify
- java.util.concurrent.ArrayBlockingQueue
  - put blocks if the queue is full
  - poll blocks if the queue is empty
  - Internally uses wait/notify

The `CopyOnWriteArrayList`

- Implements `java.util.List<E>`
- All writes to the list copy the array storing the list elements

Today: Concurrency, part 3

- The backstory
  - Motivation, goals, problems, …
- Basic concurrency in Java
  - Explicit synchronization with threads and shared memory
  - More concurrency problems
- Higher-level abstractions for concurrency
  - Data structures
  - Higher-level languages and frameworks
  - Hybrid approaches
- In the trenches of parallelism
  - Using the Java concurrency framework
  - Prefix-sums implementation

Concurrency at the language level

- Consider:

```
int sum = 0;
Iterator i = coll.iterator();
while (i.hasNext()) {
    sum += i.next();
}
```

- In python:

```
sum = 0;
for item in coll:
    sum += item
```

Parallel quicksort in Nesl

```
function quicksort(a) =
  if (#a < 2) then a
  else
   let pivot   = a[#a/2];
       lesser  = {e in a| e < pivot};
       equal   = {e in a| e == pivot};
       greater = {e in a| e > pivot};
       result  = {quicksort(v): v in [lesser,greater]};
   in result[0] ++ equal ++ result[1];
```

- Operations in {} occur in parallel
- What is the total work? What is the depth?
  - What assumptions do you have to make?

Prefix sums (a.k.a. inclusive scan)

- Goal: given array `x[0…n-1]`, compute array of the sum of each prefix of x

```
[ sum(x[0…0]),
  sum(x[0…1]),
  sum(x[0…2]),
  …
  sum(x[0…n-1]) ]
```

- e.g., `x =` `[13, 9, -4, 19, -6, 2, 6, 3]`

  prefix sums: `[13, 22, 18, 37, 31, 33, 39, 42]`

Parallel prefix sums

- Intuition: If we have already computed the partial sums `sum(x[0…3])` and `sum(x[4…7])`, then we can easily compute `sum(x[0…7])`
- e.g., `x = [13, 9, -4, 19, -6, 2, 6, 3]`

Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner

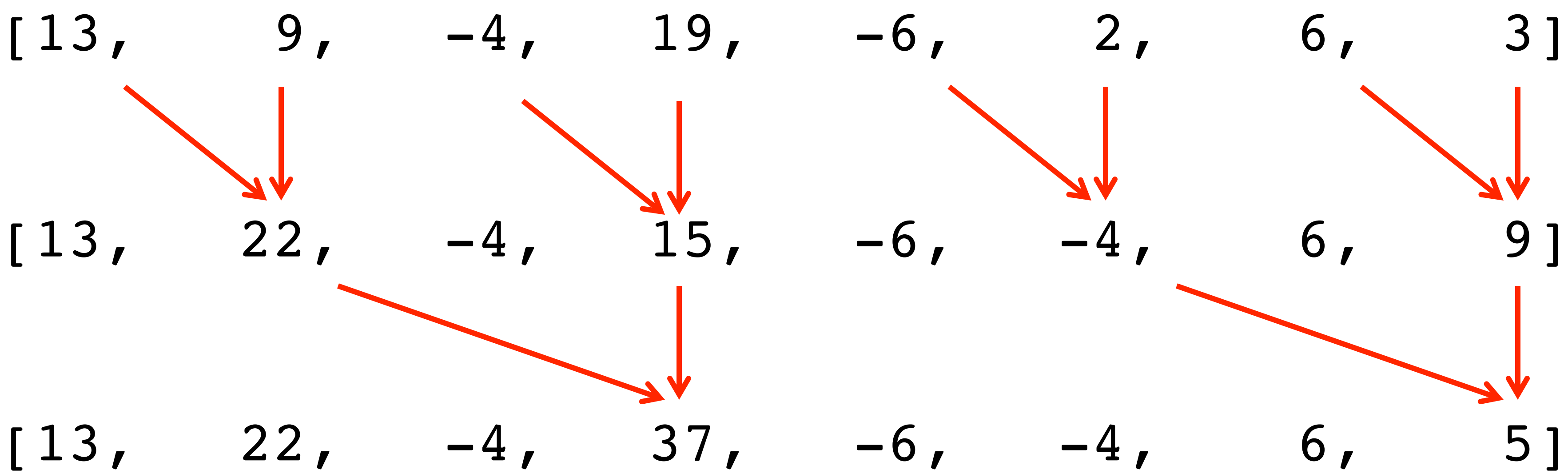
```
[13,   9,  -4,  19,  -6,   2,   6,   3]
```

```
[13,  22,  -4,  15,  -6,  -4,   6,   9]
```


institute for SOFTWARE RESEARCH

Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner



institute for SOFTWARE RESEARCH

Parallel prefix sums algorithm, winding

- Computes the partial sums in a more useful manner

```
[13,  9, -4, 19, -6,  2, 6,  3]
[13, 22, -4, 15, -6, -4, 6,  9]
[13, 22, -4, 37, -6, -4, 6,  5]
[13, 22, -4, 37, -6, -4, 6, 42]
```

⋮

Parallel prefix sums algorithm, unwinding

- Now unwinds to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]
```

```
[13,   22,   -4,   37,   -6,   33,    6,   42]
```

Parallel prefix sums algorithm, unwinding

- Now unwinds to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]

[13,   22,   18,   37,   31,   33,   39,   42]
```

- Recall, we started with:

```
[13,    9,   -4,   19,   -6,    2,    6,    3]
```

Parallel prefix sums

- Intuition: If we have already computed the partial sums `sum(x[0…3])` and `sum(x[4…7])`, then we can easily compute `sum(x[0…7])`
- e.g., `x =      [13,  9, -4, 19, -6,  2,  6,  3]`
- Pseudocode:

```
prefix_sums(x):
  for d in 0 to (lg n)-1:          // d is depth
    parallelfor i in 2^d-1 to n-1, by 2^(d+1):
      x[i+2^d] = x[i] + x[i+2^d]

  for d in (lg n)-1 to 0:
    parallelfor i in 2^d-1 to n-1-2^d, by 2^(d+1):
      if (i-2^d >= 0):
        x[i] = x[i] + x[i-2^d]
```

Parallel prefix sums algorithm, in code

- An iterative Java-esque implementation:

```
void computePrefixSums(long[] a) {
  for (int gap = 1; gap < a.length; gap *= 2) {
    parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
      a[i+gap] = a[i] + a[i+gap];
    }
  }
  for (int gap = a.length/2; gap > 0; gap /= 2) {
    parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
      a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
    }
  }
```

Parallel prefix sums algorithm, in code

- A recursive Java-esque implementation:

```
void computePrefixSumsRecursive(long[] a, int gap) {
  if (2*gap – 1 >= a.length) {
    return;
  }

  parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }

  computePrefixSumsRecursive(a, gap*2);

  parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
    a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
  }
}
```

Parallel prefix sums algorithm

- How good is this?

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See Main.java, PrefixSumsNonconcurrentParallelWorkImpl.java

Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops

```
parfor(int i=gap-1; i+gap<a.length; i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

- Partition into multiple segments, run in different threads

```
for(int i=left+gap-1; i+gap<right; i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface

```
void           run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void           start();
static void    sleep(long millis);
void           join();
boolean        isAlive();
static Thread  currentThread();
```

Recall the Java primitive concurrency tools

- The `java.lang.Runnable` interface

```
void            run();
```

- The `java.lang.Thread` class

```
Thread(Runnable r);
void            start();
static void     sleep(long millis);
void            join();
boolean         isAlive();
static Thread   currentThread();
```

- The `java.util.concurrent.Callable<V>` interface
  - Like `java.lang.Runnable` but can return a value

```
V               call();
```

A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface

```
V        get();
V        get(long timeout, TimeUnit unit);
boolean isDone();
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
```

A framework for asynchronous computation

- The `java.util.concurrent.Future<V>` interface

```
V       get();
V       get(long timeout, TimeUnit unit);
boolean isDone();
boolean cancel(boolean mayInterruptIfRunning);
boolean isCancelled();
```

- The `java.util.concurrent.ExecutorService` interface

```
Future            submit(Runnable task);
Future<V>         submit(Callable<V> task);
List<Future<V>> invokeAll(Collection<Callable<V>> tasks);
Future<V>         invokeAny(Collection<Callable<V>> tasks);
```

`Executors` for common computational patterns

- From the `java.util.concurrent.Executors` class

```
static ExecutorService newSingleThreadExecutor();
static ExecutorService newFixedThreadPool(int n);
static ExecutorService newCachedThreadPool();
static ExecutorService newScheduledThreadPool(int n);
```

- Aside: see NetworkServer.java (later)

Fork/Join: another common computational pattern

- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work

Fork/Join: another common computational pattern

- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work
- The `java.util.concurrent.ForkJoinPool` class
  - Implements `ExecutorService`
  - Executes `java.util.concurrent.ForkJoinTask<V>` or
    `java.util.concurrent.RecursiveTask<V>` or
    `java.util.concurrent.RecursiveAction`

The `RecursiveAction` abstract class

```
public class MyActionFoo extends RecursiveAction {
    public MyActionFoo(…) {
        store the data fields we need
    }

    @Override
    public void compute() {
        if (the task is small) {
            do the work here;
            return;
        }

        invokeAll(new MyActionFoo(…),  // smaller
                  new MyActionFoo(…),  // tasks
                  …);                  // …
    }
}
```

A ForkJoin example

- See PrefixSumsParallelImpl.java, PrefixSumsParallelLoop1.java, and PrefixSumsParallelLoop2.java
- See the processor go, go go!

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsSequentialImpl.java

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsSequentialImpl.java
  - n-1 additions
  - Memory access is sequential
- For PrefixSumsNonsequentialImpl.java
  - About 2n useful additions, plus extra additions for the loop indexes
  - Memory access is non-sequential
- The punchline: Constants matter.

Next week…

- Introduction to distributed systems

In-class example for parallel prefix sums

```
[7,     5,     8,  -36,    17,     2,    21,    18]
```