CSE 373

Introduction to Concurrency

reading: Grossman

slides created by Marty Stepp

http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.

Example program

- Suppose we have a program that reads several files and counts the total number of unique words in all the files combined.
  - How can we modify the program to run in parallel using threads?

```
public class WordCounts {
    public static void main(String[] args)
            throws FileNotFoundException {
        Set<String> words = new HashSet<String>();
        long startTime = System.currentTimeMillis();
        String[] files = {"bible", "abe", "hamlet", "tomsawyer"};
        for (String filename : files) {
            readFile(filename, words);
        }
        long endTime = System.currentTimeMillis();
        long elapsed = endTime - startTime;
        System.out.println("There are " + words.size()
                           + " unique words.");
        System.out.println("Took " + elapsed + " ms.");
    }
    ...
```

Example program

- Suppose we have a program that reads several files and counts the total number of unique words in all the files combined.
  - How can we modify the program to run in parallel using threads?

```
    ...
    public static void readFile(String file, Set<String> words)
            throws FileNotFoundException {
        System.out.println("Starting to read " + file + " ...");
        Scanner input = new Scanner(new File(file + ".txt"));
        while (input.hasNext()) {
            String word = input.next();
            words.add(word);
        }
        System.out.println("Done reading " + file + ".");
    }
}
```

Thread safety

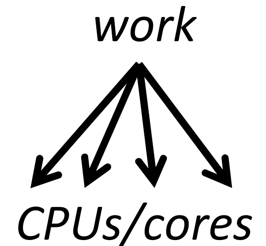
- **thread safe**: Able to be used concurrently by multiple threads.
  - Many of the Java library classes are *not* thread safe!
  - In other words, if two threads access the same object, things break.

- Examples:
  - `ArrayList` and other collections from `java.util` are not thread safe; two threads changing the same list at once may break it.
  - `StringBuilder` is not thread safe.
  - Java GUIs are not thread safe; if two threads are modifying a GUI simultaneously, they may put the GUI into an invalid state.

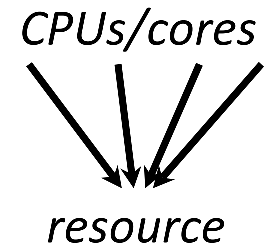
- Counterexamples:
  - The `Random` class chooses numbers in a thread-safe way.
  - Some input/output (like `System.out`) is thread safe.

Parallel vs. concurrent

- **parallel:** Using multiple processing resources (CPUs, cores) at once to solve a problem faster.
  - Example: A sorting algorithm that has several threads each sort part of the array.



- **concurrent:** Multiple execution flows (e.g. threads) accessing a shared resource at the same time.
  - Example: Many threads trying to make changes to the same data structure (a global list, map, etc.).



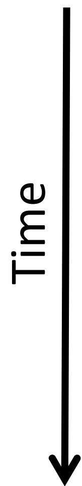
Concurrency

- Unlike parallelism, not always about running faster.
  - Even a single-CPU, single-core machine may want concurrency.

- Useful for:
  - *App responsiveness*
    - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
  - *Processor utilization* (mask I/O latency)
    - If 1 thread is stuck working, others have something else to do
  - *Failure isolation*
    - Convenient structure if want to interleave multiple tasks and do not want an exception in one to stop the other

Time slicing

- If a given piece of code is run by two threads at once:
  - The order of which thread gets to run first is unpredictable.
  - How many statements of one thread run before the other thread runs some of its own statements is unpredictable.




```
// Thread 1
public void foo() {
    ...
    statement1;
    statement2;
    ...
    statement3;
}
```

```
// Thread 2
public void foo() {
    statement1;
    ...
    ...
    statement2;
    ...
    statement3;
}
```

Thread-unsafe code

- How can the following class be broken by multiple threads?

```
1  public class Counter {
2    private int c = 0;
3    public void increment() {
4      int old = c;
5      c = old + 1;  // c++;
6    }
7    public void decrement() {
8      int old = c;
9      c = old - 1;  // c--;
10   }
11   public int value() {
12     return c;
13   }
14 }
```

Scenario that breaks it:

- Threads A and B start.
- A calls `increment` and runs to the end of line 4. It retrieves the `old` value of 0.
- B calls `decrement` and runs to the end of line 8. It retrieves the `old` value of 0.
- A sets c to its `old` (0) + 1.
- B sets c to its `old` (0) - 1.
- The final `value()` is -1, though after one increment and one decrement, it should be 0!

Synchronized blocks

```
// synchronized block:
// uses the given object as a lock
synchronized (object) {
    statement(s);
}
```

- Every Java object can act as a "lock" for concurrency.
  - A thread T_1 can ask to run a block of code, "synchronized" on a given object *O*.
    - If no other thread is using *O*, then T_1 locks the object and proceeds.
    - If another thread T_2 is already using *O*, then T_1 becomes blocked and must wait until T_1 is finished using O. Then T_1 can proceed.

Synchronized methods

```
// synchronized method: locks on "this" object
public synchronized type name(parameters) { ... }

// synchronized static method: locks on the given class
public static synchronized type name(parameters) { ... }
```

- A synchronized method grabs the object or class's lock at the start, runs to completion, then releases the lock.
  - A shorthand for wrapping the entire body of the method in a `synchronized (this) {...}` block.
  - Useful for methods whose entire bodies should not be entered by multiple threads at the same time.

```
public synchronized void readFile(String name) {...}
```

Synchronized counter

```
public class Counter {
    private int c = 0;

    public synchronized void increment() {
        int old = c;
        c = old + 1;  // c++;
    }

    public synchronized void decrement() {
        int old = c;
        c = old - 1;  // c--;
    }

    public int value() {
        return c;
    }
}
```

- Should the `value` method be synchronized? Why/why not?

Critical section problem

- **critical section**: A piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread.
  - Multiple threads write to the set, so its state can become corrupted.
  - Inside the `HashSet`'s `add` method looks something like this:

Time ↓

```
// Thread 1:  add(42);
public void add(E value) {
  int h = hash(value);  // 2
  Node n = new Node(value);
  n.next = elements[h];
  ...
  ...
  ...
  ...
  elements[h] = n;
  size++;
}
```

```
  ...
  ...
  ...
  ...
// Thread 2:  add(72);
public void add(E value) {
  int h = hash(value);  // 2
  Node n = new Node(value);
  n.next = elements[h];
  ...
  ...
  elements[h] = n;  // 42 lost
  size++;
}
```

Concurrent word counter

- What happens to our word counter if we make the `readFile` method `synchronized`?

```
    ...
    public static synchronized void readFile(
            String file, Set<String> words)
                    throws FileNotFoundException {
        System.out.println("Starting to read " + file + " ...");
        Scanner input = new Scanner(new File(file + ".txt"));
        while (input.hasNext()) {
            String word = input.next();
            words.add(word);
        }
        System.out.println("Done reading " + file + ".");
    }
}
```

  - The program essentially becomes sequential; no two threads are able to read their files at the same time.

Better concurrent counter

- A version that is better because of a smaller critical section:

```
    ...
    public static void readFile(String file, Set<String> words)
                    throws FileNotFoundException {
        System.out.println("Starting to read " + file + " ...");
        Scanner input = new Scanner(new File(file + ".txt"));
        while (input.hasNext()) {
            String word = input.next();
            synchronized (words) {
                words.add(word);
            }
        }
        System.out.println("Done reading " + file + ".");
    }
}
```

  - The program is better than the previous one, but not much speed-up is seen from the multiple threads. Why not?

Synchronized collections

- Java provides thread-safe collection *wrappers* via static methods in the `Collections` class:

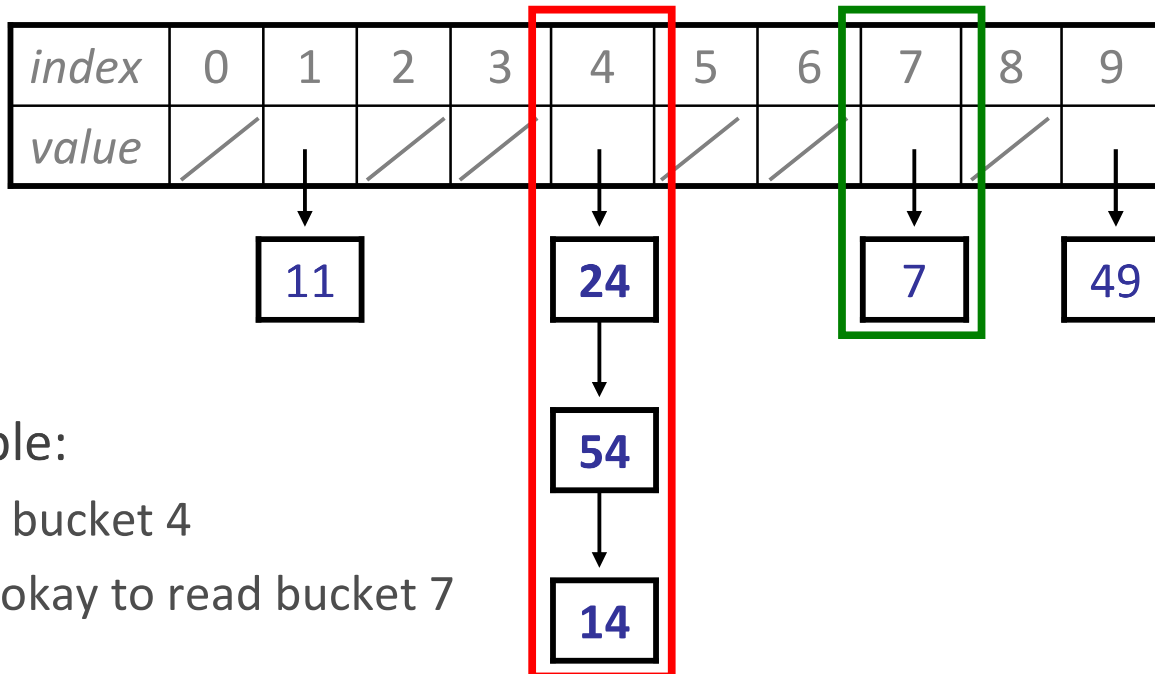
Method
`Collections.synchronizedCollection(`**coll**`)`
`Collections.synchronizedList(`**list**`)`
`Collections.synchronizedMap(`**map**`)`
`Collections.synchronizedSet(`**set**`)`

```
Set<String> words = new HashSet<String>();
words = Collections.synchronizedSet(words);
```

  - These are essentially the same as wrapping each operation on the collection in a `synchronized` block.
    - Simpler, but not more efficient, than the preceding code.

Fine-grained critical sections

- Technically, the shared resource is the linked list in the given hash bucket `elements[h]`, not the *entire* hash table array.
  - What if the set could lock just that bucket?
  - If my thread is updating bucket *h*, other threads that wanted to read/write other hash buckets besides *h* could still do so.



  - Example:
    - lock bucket 4
    - still okay to read bucket 7

Fine-grained locking

```
// keep a lock for each bucket

// Thread 1:  add(42);
public void add(E value) {
  int h = hash(value);  // 2
  Node n = new Node(value);
  ...
  ...
  ...
  ...
  synchronized (locks[h]) {
    n.next = elements[h];
    elements[h] = n;
    size++;
  }
}
```

```
    ...
    ...
    ...
    ...
    ...
// Thread 2:  add(72);
public void add(E value) {
  int h = hash(value);  // 2
  Node n = new Node(value);
  ...
  synchronized (locks[h]) {
    ... blocked ...
    ... blocked ...
    ... blocked ...
    n.next = elements[h];
    elements[h] = n;
    size++;
  }
}
```

Time

Concurrent collections

- New package `java.util.concurrent` contains collections that are optimized to be safe for use by multiple threads:
  - class **ConcurrentHashMap**<K, V> implements Map<K, V>
  - class **ConcurrentLinkedDeque**<E> implements Deque<E>
  - class **ConcurrentSkipListSet**<E> implements Set<E>
  - class **CopyOnWriteArrayList**<E> implements List<E>

- These classes are generally faster than using a synchronized version of the normal collections because multiple threads are actually able to use them at the same time, to a degree.
  - hash map: one thread in each hash bucket at a time
  - deque: one thread modifying each end of the deque (front/back)
  - ...

Object lock methods

- *Every* Java object has a built-in internal "lock".
  - A thread can "wait" on an object's lock, causing it to pause.
  - Another thread can "notify" on an object's lock, unpausing any other thread(s) that are currently waiting on that lock.
  - An implementation of *monitors*, a classic concurrency construct.

method	description
`notify``()`	unblocks one random thread waiting on this object's lock
`notifyAll``()`	unblocks all threads waiting on this object's lock
`wait``()` **`wait`**`(`**`ms`**`)`	causes the current thread to wait (block) on this object's lock, indefinitely or for a given # of ms

  - These methods are not often used directly; but they are used internally by other concurrency constructs (see next slide).

The volatile keyword

```
private volatile type name;
```

- **volatile field**: An indication to the VM that multiple threads may try to access/update the field's value at the same time.
  - Causes Java to immediately flush any internal caches any time the field's value changes, so that later threads that try to read the value will always see the new value (never the stale old value).
  - Allows limited safe concurrent access to a field inside an object even if another thread may modify the field's value.
  - Does not solve all concurrency issues; should be replaced by `synchronized` blocks if more complex access is needed.

Deadlock

- **liveness**: Ability for a multithreaded program to run promptly.
- **deadlock**: Situation where two or more threads are blocked forever, waiting for each other.
  - Example: Each is waiting for the other's locked resource.
  - Example: Each has too large of a `synchronized` block.
- **livelock**: Situation where two or more threads are caught in an infinite cycle of responding to each other.
- **starvation**: Situation where one or more threads are unable to make progress because of another "greedy" thread.
  - Example: thread with a long-running synchronized method

New classes for locking

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

Class/interface	description
`Lock`	an interface for controlling access to a shared resource
`ReentrantLock`	a class that implements `Lock`
`ReadWriteLock`	like `Lock` but separates read operations from writes
`Condition`	a particular shared resource that can be waited upon; conditions are acquired by asking for one from a `Lock`

- These classes offer higher granularity and control than the `synchronized` keyword can provide.
  - Not needed by most programs.
  - `java.util.concurrent` also contains blocking data structures.