Practical Concurrent and Parallel Programming 8

Peter Sestoft
IT University of Copenhagen

Friday 2014-10-24
Plan for today

• Graphical user interface toolkits, eg Swing
  – not thread-safe, access from event thread only
• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated
• A thread-based lift simulator with GUI
• Atomic long with “thread striping” (week 7)
• Shared mutable data on multicore is slow
GUI toolkits are single-threaded

• Java Swing components are **not** thread-safe
  – This is intentional
  – Ditto .NET’s System.Windows.Forms and others

• Multithreaded GUI toolkits
  – are difficult to use
  – deadlock-prone, because actions are initiated both
    • *top-down*: from user towards operating system
    • *bottom-up*: from operating system to user interface
    • locking in different orders ... hence deadlock risk

• In Swing, at least two threads:
  – Main Thread – runs `main(String[] args)`
  – Event Thread – runs ActionListeners and so on
From Graham Hamilton’s blog post “Multithreaded toolkits: A failed dream?”

- “In general, GUI operations start at the top of a stack of library abstractions and go "down". I am operating on an abstract idea in my application that is expressed by some GUI objects, so I start off in my application and call into high-level GUI abstractions, that call into lower level GUI abstractions, that call into the ugly guts of the toolkit, and thence into the OS.
- In contrast, input events start off at the OS layer and are progressively dispatched "up" the abstraction layers, until they arrive in my application code.
- Now, since we are using abstractions, we will naturally be doing locking separately within each abstraction.
- And unfortunately we have the classic lock ordering nightmare: we have two different kinds of activities going on that want to acquire locks in opposite orders. So deadlock is almost inevitable.” (19 October 2004)

https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html
Java Swing GUI toolkit dogmas

- **Dogma 1**: “Time-consuming tasks should not be run on the Event Thread”
  - Otherwise the application becomes unresponsive

- **Dogma 2**: “Swing components should be accessed on the Event Thread only”
  - The components are not thread-safe

- But if another thread does long-running work, how can it show the results on the GUI?
  - Define the work in SwingWorker subclass instance
  - Use `execute()` to run it on a worker thread
  - The Event Thread can pick up the results
A short computation on the event thread

- Main thread may create GUI components
  - But should not change eg. background color later
- Event thread calls the ActionListener
  - And can change the background color

```java
final JFrame frame = new JFrame("TestButtonGui");
final JPanel panel = new JPanel();
final JButton button = new JButton("Press here");
frame.add(panel);
panel.add(button);
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        panel.setBackground(new Color(random.nextInt()));
    }
});
frame.pack(); frame.setVisible(true);
```
Main thread and event thread

Main thread

create button
create panel

button

panel

addActionListener

event thread

ActionPerformed

setBackg

paint(g)

repaint()

Click
Using the main thread for blinking

```java
final JPanel panel = new JPanel() {
    public void paint(Graphics g) {
        super.paint(g);
        if (showBar) {
            g.setColor(Color.RED);
            g.fillRect(0, 0, 10, getHeight());
        }
    }
};
final JButton button = ...
frame.pack(); frame.setVisible(true);
while (true) {
    try { Thread.sleep(800); } // milliseconds
    catch (InterruptedException exn) { }
    showBar = !showBar;
    panel.repaint();
}
```

- `repaint()` may be called by any thread
- Causes event thread to call `repaint(g)` later
Fetching webpages on event thread

```java
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for (String url : urls) {
            System.out.println("Fetching " + url);
            String page = getPage(url, 200);
            textArea.append(String.format(..., url, page.length()));
        }
    }
});
```

- Occupies event thread for many seconds
  - The GUI is unresponsive in the meantime
  - Results not shown as they become available
    - GUI gets updated only after all fetches
  - Cancellation would not work
    - Cancel button event processed only after all fetches
  - A progress bar would not work
    - Gets updated only after all fetches
static class DownloadWorker extends SwingWorker<String, String> {
    private final TextArea textArea;

    public String doInBackground() {
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
            result = String.format("%-40s%7d%n", url, page.length());
            sb.append(result);
        }
        return sb.toString();
    }

    public void done() {
        try { textArea.append(get()); } 
        catch (InterruptedException exn) { }
        catch (ExecutionException exn) { throw new RuntimeException(...); }
    }
}

• SwingWorker<T,V> implements Future<T>
• .NET has System.ComponentModel.BackgroundWorker
Fetching web with SwingWorker

```java
DownloadWorker downloadTask = new DownloadWorker(textArea);
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.execute();
    }
});
```

- Event thread runs `execute()`
- Worker thread runs `doInBackground()`
  - which returns the full result when computed
- Event thread runs `done()`
  - obtains the already-computed result with `get()`
  - and writes the result to the `textArea`
Worker thread and event thread

Worker thread

```
Worker thread

downloadTask
downloadTask.execute()
downloadTask

textContent
doInBackground()
getPage() "A"
getPage() "B"
getPage() "C"

"A B C"

done()

append("A B C")
```

Event thread
Add progress notification

```java
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        int count = 0;
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
            result = String.format("%-40s%7d%n", url, page.length());
            sb.append(result);
            setProgress((100 * ++count) / urls.length);
        }
        return sb.toString();
    }
}
```

- In the GUI setup, add:

```java
downloadTask.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        if ("progress".equals(e.getPropertyName())) {
            progressBar.setValue((Integer)e.getNewValue());
        }
    }
});
```
Add cancellation

```java
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        for (String url : urls) {
            if (isCancelled())
                break;
            ...
            sb.append(result);
        }
        return sb.toString();
    }
    public void done() {
        try { textArea.append(get()); }
        catch (InterruptedException exn) { }
        catch (ExecutionException exn) { throw new RuntimeExc...; }
        catch (CancellationException exn) { textArea.append("Yrk"); }
    }
}
```

- In the GUI setup, add:

```java
cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.cancel(false);
    }});
```
Progress and cancellation

Worker thread

downloadTask
downloadTask.execute()
doInBackground()
getPage() “A”
setProgress(…)
getPage() “B”
setProgress(…)

TextArea

event thread

progressBar
.setProgress(…)
cancel(false)
done()
append(“Yrk”)
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        for (String url : urls) {
            String page = getPage(url, 200),
                result = String.format("%-40s%7d\n", url, page.length());
            publish(result);
        }
    }

    public void process(List<String> results) {
        for (String result : results)
            textArea.append(result);
    }
}

- Worker thread calls `publish(...)` a few times
- Event thread calls `process` with results from calls to `publish` since last call to `process`
Event thread and downloadTask

Worker thread

downloadTask

textArea

event thread

doInBackground() → getPage() → publish(“A”)

doInBackground() → getPage() → publish(“B”)

doInBackground() → process([“A”, “B”]) → append(“A”)

doInBackground() → process([“A”, “B”]) → append(“B”)

doInBackground() → getPage() → publish(“C”)

doInBackground() → process([“C”]) → append(“C”)

doInBackground() → process([“C”]) → done()

doInBackground() → process([“C”]) → append(“”)

downloadTask.execute()
SwingUtilities static methods

- May be called from any thread:
  - boolean isEventDispatchThread()
    • True if executing thread is the Event Thread
  - void invokeLater(Runnable cmd)
    • Execute cmd.run() asynchronously on the Event Thread
  - void invokeAndWait(Runnable command)
    • Execute cmd.run() on the Event Thread, wait to complete

- SwingWorker = these + Java executors
  - Goetz Listings 9.2 and 9.7 indicate how

- Other methods that any thread may call:
  - adding and removing listeners on components
    • but the listeners are *called* only on the Event Thread
  - comp.repaint() and comp.revalidate()
Avoids interaction with a partially constructed GUI
- because the Event Thread is busy constructing the GUI

```java
public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            final Random random = new Random();
            final JFrame frame = new JFrame("TestButtonGui");
            final JPanel panel = new JPanel();
            final JButton button = new JButton("Press here");
            frame.add(panel);
            panel.add(button);
            button.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    panel.setBackground(new Color(random.nextInt()));
                }
            });
            frame.pack(); frame.setVisible(true);
        }
    });
}
```
Plan for today

• Graphical user interface toolkits, eg Swing
  – not thread-safe, access from event thread only
• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated
• A thread-based lift simulator with GUI
• Atomic long with “thread striping” (week 7)
• Shared mutable data on multicore is slow
Example: 2 lifts, 7 floors, 26 buttons

Concurrent:
- Two lift threads + the event thread
- Inside-lift buttons
- Outside-lift buttons

Two lift cages move 26 buttons pressed
Modeling and visualizing the lifts

• Use event thread for buttons (obviously)
  – Inside requests: *this lift* must go to floor n
  – Outside requests: *some lift* must go to floor n, and then up (or down)

• An object for each lift
  – to hold current floor, and floors yet to be visited
  – to compute time to serve an outside request

• A thread for each lift
  – to update its state 16 times a second
  – to cause the GUI to display it

• A controller object
  – to decide which lift should serve an outside request
The lift simulator GUI

- Lift 1 shaft
- Buttons inside lift 1
- Lift 1 cage
- Buttons outside lifts
Lift controller algorithm

- When outside button Up on floor n is pressed
  - Ask each lift how long it would take to get to floor n while continuing up afterwards
  - Then order the fastest lift to serve floor n

```java
class LiftController {
    private final Lift[] lifts;
    ...
    public void someLiftTo(int floor, Direction dir) {
        double bestTime = Double.POSITIVE_INFINITY;
        int bestLift = -1;
        for (int i=0; i<lifts.length; i++) {
            double thisLiftTime = lifts[i].timeToServe(floor, dir);
            if (thisLiftTime < bestTime) {
                bestTime = thisLiftTime;
                bestLift = i;
            }
        }
        lifts[bestLift].customerAt(floor, dir);
    }
}
```
The state of a lift

- Current floor and direction (None, Up, Down)
- required stops and directions, \texttt{stops[floor]}:
  - \texttt{null}: no need to stop at this floor
  - \texttt{None}: stop, don’t know future direction
  - \texttt{Down}: stop, then continue down
  - \texttt{Up}: stop, then continue up
  - \texttt{Both}: stop, then up (down); stop, then down (up)

```java
class Lift implements Runnable {
    private double floor;
    private Direction direction;   // None or Up or Down
    // @GuardedBy("this")
    private final Direction[] stops;
    ...
    public synchronized void customerAt(int floor, Direction thenDir) {
        setStop(floor, thenDir.add(getStop(floor)));
    }
}
```

- Accessed only on lift thread
- Called by controller
The lift’s behavior when going Up

• If at a floor, check whether to stop here
  – If so, open+close doors and clear from stops table
• If not yet at highest requested stop
  – move up a bit and refresh display
  – otherwise stop moving

```java
switch (direction) {
    case Up:
        if ((int)floor == floor) { // At a floor, maybe stop here
            Direction afterStop = getStop((int)floor);
            if (afterStop != null && (afterStop != Down || (int)floor == highestStop())) {
                openAndCloseDoors();
                subtractFromStop((int)floor, direction);
            }
        }
        if (floor < highestStop()) {
            floor += direction.delta / steps;
            shaft.moveTo(floor, 0.0);
        } else
            direction = Direction.None;
    break;
    case Down: ... similar to Up ...
    case None: ... if any stops[floor] != null, start moving in that direction ...
}
```
Lift GUI thread safety

• Dogma 1, no long-running on event thread:
  – `sleep()` happens on lift threads, not event thread

• Dogma 2, only event thread works on GUI:
  – Lift thread calls `shaft.moveTo`,
  – which calls `repaint()`,
  – so event thread calls `paint(g)`, OK

• Lift and event threads access `stops[]` array
  – guarded by lift instance `this`

• Only lift thread accesses `floor` and `direction`
  – not guarded
Lift modeling reflection

- Seems reasonable to have a thread per lift
  - because they move concurrently
- Why not a thread for the controller?
  - because activated only by the external buttons
  - but what about supervising the lifts? e.g. if the lift sent to floor 4 going Up gets stuck at floor 3 by some fool with a lot of boxes?
- In Erlang, with message-passing, use
  - a “process” (task) for each lift
  - a “process” (task) for each floor, a “local controller”
  - no central controller

- Also Akka library, week 13-14

Plan for today

- Graphical user interface toolkits, eg Swing
  - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
  - Progress bar
  - Cancellation
  - Display results as they are generated
- A thread-based lift simulator with GUI
- **Atomic long with “thread striping”** (wk 7)
- Shared mutable data on multicore is slow
A “striped” thread-safe long

• Use case: more writes (\texttt{add}) than reads (\texttt{get})

• Vastly different scalability
  - (a) Java 5’s AtomicLong
  - (b) Java 8’s LongAdder
  - (c) Home-made synchronized LongCounter
  - (d) Home-made striped long using AtomicLongArray
  - (e) Home-made striped long with scattered allocation

• Ideas
  - (d,e) Use thread’s hashCode to reduce update collisions
  - (e) Scatter AtomicLongs to avoid false cache line sharing

<table>
<thead>
<tr>
<th></th>
<th>i7 4c</th>
<th>AMD 32c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>942</td>
<td>3011</td>
</tr>
<tr>
<td>(b)</td>
<td>65</td>
<td>54</td>
</tr>
<tr>
<td>(c)</td>
<td>1450</td>
<td>14921</td>
</tr>
<tr>
<td>(d)</td>
<td>427</td>
<td>1611</td>
</tr>
<tr>
<td>(e)</td>
<td>108</td>
<td>922</td>
</tr>
</tbody>
</table>

Wall clock time (ms) for 32 threads making 1 million additions each
Dividing a long into 32 “stripes”

```java
class NewLongAdder {
    private final static int NSTRIPES = 32;
    private final AtomicLongArray counters = new AtomicLongArray(NSTRIPES);

    public void add(long delta) {
        counters.addAndGet(Thread.currentThread().hashCode() % NSTRIPES, delta);
    }

    public long longValue() {
        long result = 0;
        for (int stripe=0; stripe<NSTRIPES; stripe++)
            result += counters.get(stripe);
        return result;
    }
}
```

- Two threads unlikely to add to same stripe
- Each stripe has thread-affinity
  - if accessed by thread, likely to be accessed again
- So, fast despite the cost of `hashCode()`
Plan for today

• Graphical user interface toolkits, eg Swing
  – not thread-safe, access from event thread only
• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated
• A thread-based lift simulator with GUI
• An atomic long with “thread striping” (week 7)
• Shared mutable data on multicore is slow
A typical multicore CPU with three levels of cache

- Floating-point register add or mul: 0.4 ns
- RAM access: > 100 ns
Fix 1: Each processor core has a cache

- Cache = simple hardware hashtable
- Stores recently accessed values from RAM
- Cache is much faster than RAM

- Two caches may have different values for a given memory address
Fix 2: Get all caches to agree

- Cache coherence; cache line state = M,E,S,I

<table>
<thead>
<tr>
<th>State</th>
<th>Cache line</th>
<th>Excl</th>
<th>RAM</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Modified by me</td>
<td>Y</td>
<td>not OK</td>
<td>from cache</td>
<td>to cache</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Not modified</td>
<td>Y</td>
<td>OK</td>
<td>from cache</td>
<td>to cache -&gt; M</td>
</tr>
<tr>
<td>Shared</td>
<td>Others have it too</td>
<td>N</td>
<td>OK</td>
<td>send invalidate</td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>Not in use by me</td>
<td>-</td>
<td>-</td>
<td>elsewhere</td>
<td>send invalidate</td>
</tr>
</tbody>
</table>

- A cache line
  - has 4 states
  - and 12 transitions a-l

- Cache messages
  - sent by cores to others
  - via memory bus
  - to make caches agree
Fast and slow cache cases

- The cache is fast when
  - the local core “owns” the data (state M or E), or
  - data is shared (S) but local core only reads it

- The cache is slow when
  - the data is shared (S) and we want to write it, or
  - the data is not in cache (I)
    - possibly because “owned” by another core

<table>
<thead>
<tr>
<th></th>
<th>This core wants to</th>
<th>Messages</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unshared mutable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M M Read cache line</td>
<td>0</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td>M M Write cache line</td>
<td>0</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td>E E Read cache line</td>
<td>0</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td>E M Write cache line</td>
<td>0</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td>Shared immutable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S S Read cache line</td>
<td>0</td>
<td>fast</td>
<td></td>
</tr>
<tr>
<td>I S Read cache line</td>
<td>1+1</td>
<td>slow</td>
<td></td>
</tr>
<tr>
<td>Shared mutable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S M Write cache line</td>
<td>1+N</td>
<td>very slow</td>
<td></td>
</tr>
<tr>
<td>I M Write cache line</td>
<td>1+1+N</td>
<td>very slow</td>
<td></td>
</tr>
</tbody>
</table>
## Transitions and messages

A write in a non-exclusive state requires acknowledge ack* from all other cores

Shared mutable state is slow on big machines

### Table

<table>
<thead>
<tr>
<th></th>
<th>Cause</th>
<th>I send</th>
<th>I receive</th>
<th>My response</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>a (Send update to RAM)</td>
<td>writeback</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>b Write</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>c Other wants to write</td>
<td>-</td>
<td>read inv</td>
<td>read resp, inv ack</td>
</tr>
<tr>
<td>I</td>
<td>d Atomic read-mod-write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>e Atomic read-mod-write</td>
<td>read inv</td>
<td>inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>f Other wants to read</td>
<td>-</td>
<td>read</td>
<td>read resp</td>
</tr>
<tr>
<td>E</td>
<td>g Other wants to read</td>
<td>-</td>
<td>read</td>
<td>read resp</td>
</tr>
<tr>
<td>S</td>
<td>h Will soon write</td>
<td>inv</td>
<td>inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>i Other wants atomic rw</td>
<td>-</td>
<td>read inv</td>
<td>read resp, inv ack</td>
</tr>
<tr>
<td>I</td>
<td>j Want to write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>I</td>
<td>k Want to read</td>
<td>read</td>
<td>read resp</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>l Other wants to write</td>
<td>-</td>
<td>inv</td>
<td>inv ack 37</td>
</tr>
</tbody>
</table>

### Diagram

Transitions and messages diagram with nodes labeled M, E, I, and S. Edges between nodes indicate transitions and messages.
One more performance problem: “false sharing” because of cache lines

- A cache line typically is 32 bytes
  - gives better memory bus utilization
  - prefetches data (in array) that may be needed next

- Thus invalidating one (8 byte) long may invalidate the neighboring 3 longs!

- Frequently written memory locations should not be on the same cache line

- Attempts to fix this by “padding”
  - may look very silly (next slide)
  - are not guaranteed to help
  - yet are used in the Java class library code
Scattering the stripes of a long

```java
class NewLongAdderPadded {
    private final static int NSTRIPES = 32;
    private final AtomicLong[] counters;

    public NewLongAdderPadded() {
        this.counters = new AtomicLong[NSTRIPES];
        for (int stripe=0; stripe<NSTRIPES; stripe++) {
            // Believe it or not, this sometimes speeds up the code,
            // presumably because avoids false sharing of cache lines:
            new Object(); new Object(); new Object(); new Object();
            counters[stripe] = new AtomicLong();
        }
    }
}
```

- Allocate many AtomicLongs
  - instead of AtomicLongArray
- Scatter the AtomicLongs
  - by allocating some Objects in between

Avoid side-by-side AtomicLong allocation unless JVM is too clever
This week

• Reading this week
  – Goetz et al chapter 9
  – Optional: McKenney: *Memory barriers*

• Exercises week 8 = mandatory hand-in 4
  – The week 7 exercises: Write well-performing and scalable software using lock striping, immutability, Java atomics, and visibility rules
  – You can write responsive and correct user interfaces

• Read before next week’s lecture
  – Goetz chapter 12