

Practical Concurrent and Parallel Programming 7

Matteo Ceccarelo
IT University of Copenhagen

Friday 2018-10-12

Plan for today

- **Threadsafe long integers with “striping”**
- Memory, cache and cache coherence
 - Shared mutable data on multicore is slow
- 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

Based on slides by
Peter Sestoft

Flashback week 6: Maintaining hashmap size information

- Using a single AtomicLong limits scalability
- We used one size component per stripe:

```
class StripedMap<K,V> implements OurMap<K,V> {  
    private final int[] sizes;  
    public int size() {  
        ... for-loop summing sizes[i] ...  
    }  
    ...  
}
```

- Fast updates but slow `size()` queries
 - *Very slow* if millions of stripes, as in Java 8 CHM
- Java 8 ConcurrentHashMap uses a LongAdder
 - A `long` counter that scales well with many threads
 - If there are more writes (`add`) than reads (`get`)

Dividing a long into 31 "stripes"

Thread's hashCode
selects stripe

```
class NewLongAdder {
    private final static int NSTRIPES = 31;
    private final AtomicLong[] counters;
    ...
    public void add(long delta) {
        counters[Thread.currentThread().hashCode() % NSTRIPES].addAndGet(delta);
    }

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

- Two threads unlikely to add to same stripe
- Each thread has a home stripe – "affinity"
 - if accessed by thread, likely to be accessed again
- So, quite fast despite the cost of **hashCode()**

Thread-safe longs: simple, striped, ...

- Vastly different scalability
 - (a) Java 5's AtomicLong
 - (b) Home-made single-lock LongCounter (week 1)
 - (c) Home-made striped long using AtomicLongArray
 - (d) Home-made striped long using AtomicLong[]
 - (e) Home-made striped long with scattered allocation
 - (f) Java 8's LongAdder

- Ideas

- (c,d,e) Use thread's hashCode to reduce update collisions
- (e) Scatter AtomicLongs to avoid false cache line sharing
- Difference d-e is scattering of AtomicLong objects in memory

	i7 4c	AMD 32c	Xeon 48c
(a)	974	3011	667
(b)	499	14921	815
(c)	422	1611	956
(d)	183	-	296
(e)	114	922	135
(f)	64	54	22

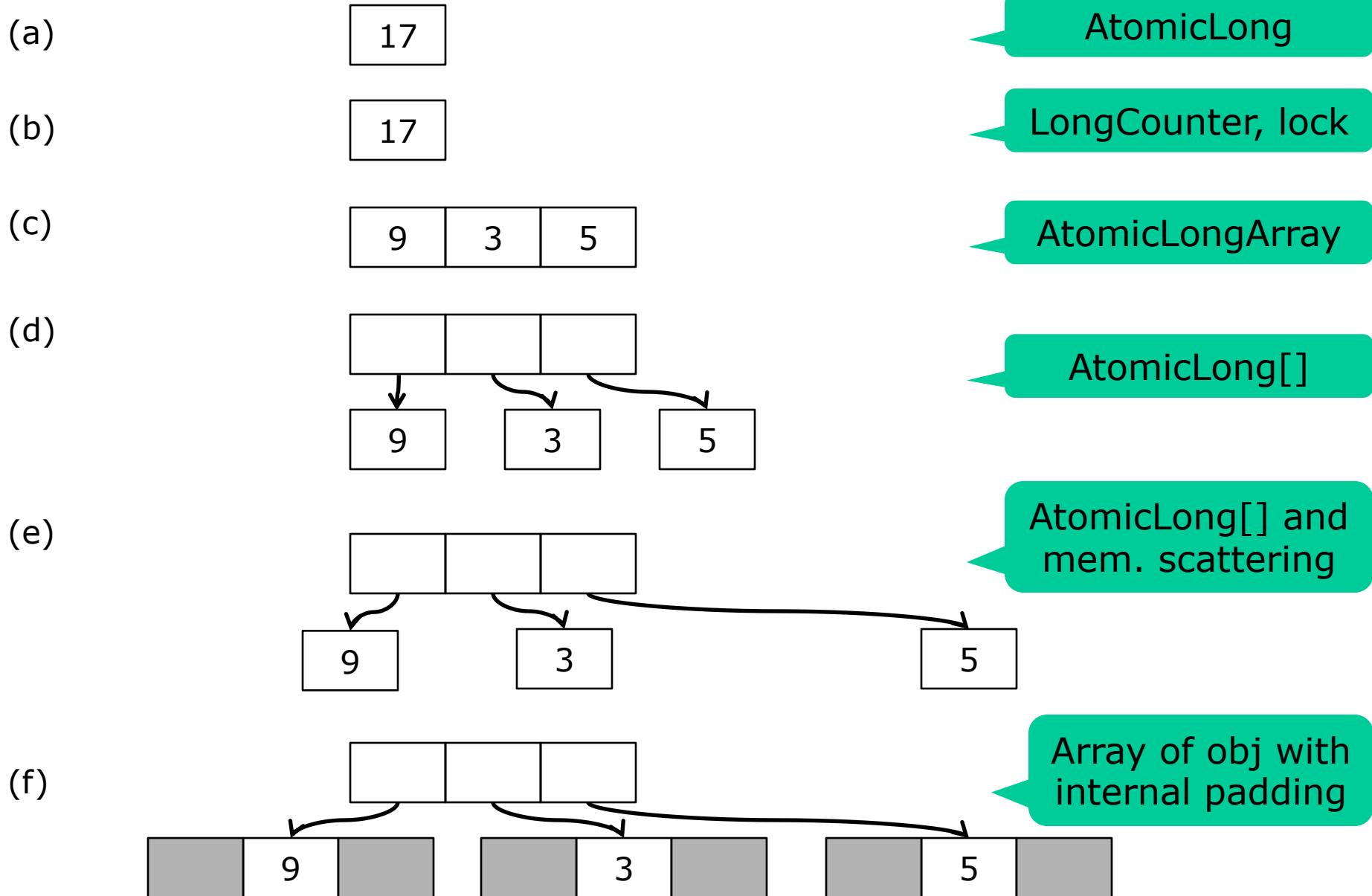
Thread-safe longs: simple, striped, ...

- Vastly different scalability
 - (a) Java 5's AtomicLong
 - (b) Home-made single-lock LongCounter (week 1)
 - (c) Home-made striped long using AtomicLongArray
 - (d) Home-made striped long using AtomicLong[]
 - (e) Home-made striped long with scattered allocation
 - (f) Java 8's LongAdder

- Ideas
 - (c,d,e) Use thread's hashCode to reduce update collisions
 - (e) Scatter AtomicLongs to avoid false cache line sharing
 - Difference d-e is scattering of AtomicLong objects in memory

	i7 4c	AMD 32c	Xeon 48c
(a)	974	3011	667
(b)	499	14921	815
(c)	422	1611	956
(d)	183	-	296
(e)	114	922	135
(f)	64	54	22

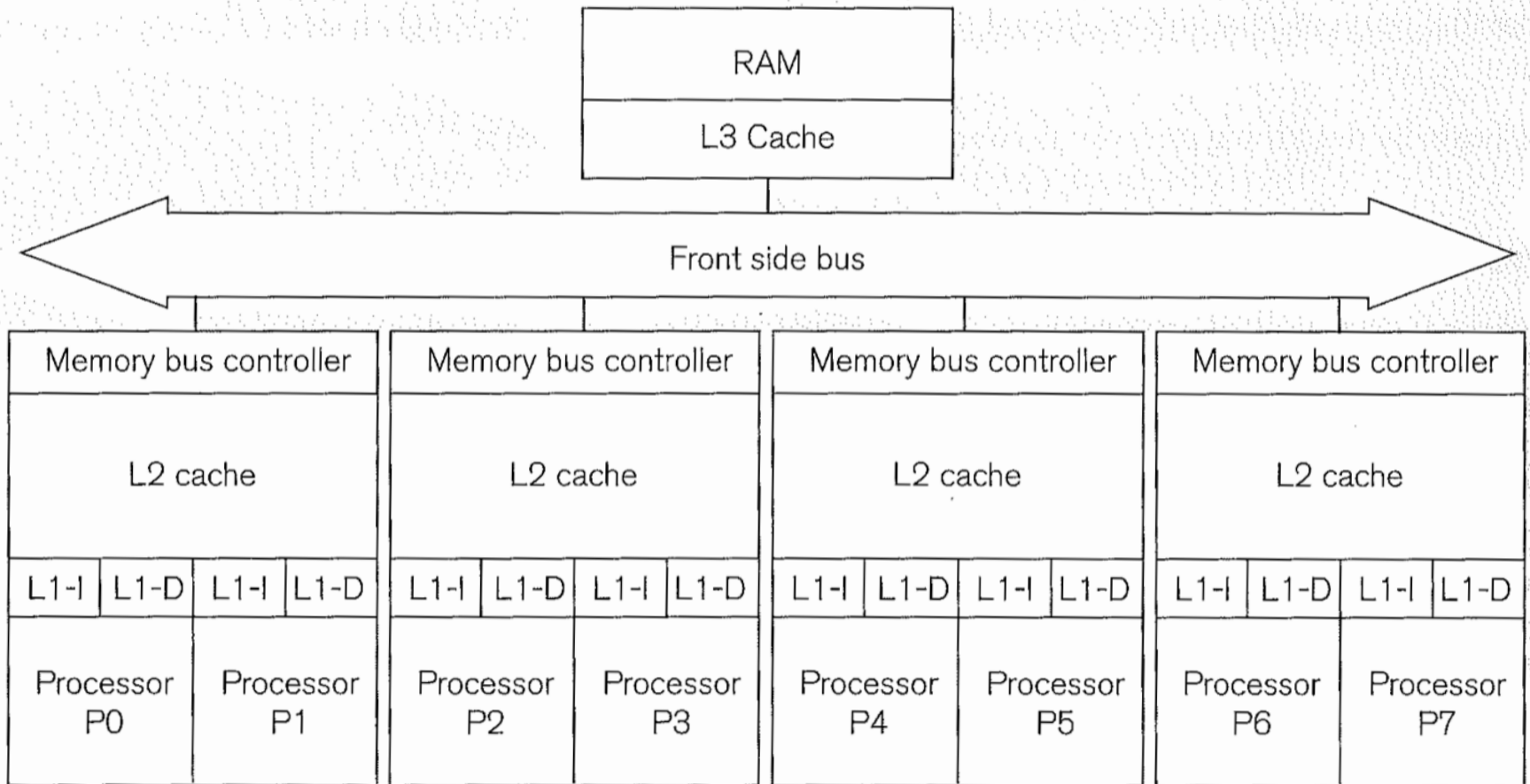
The long adders' memory layouts



Plan for today

- Threadsafe long integers with "striping"
- **Memory, cache and cache coherence**
 - Shared mutable data on multicore is slow
- 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

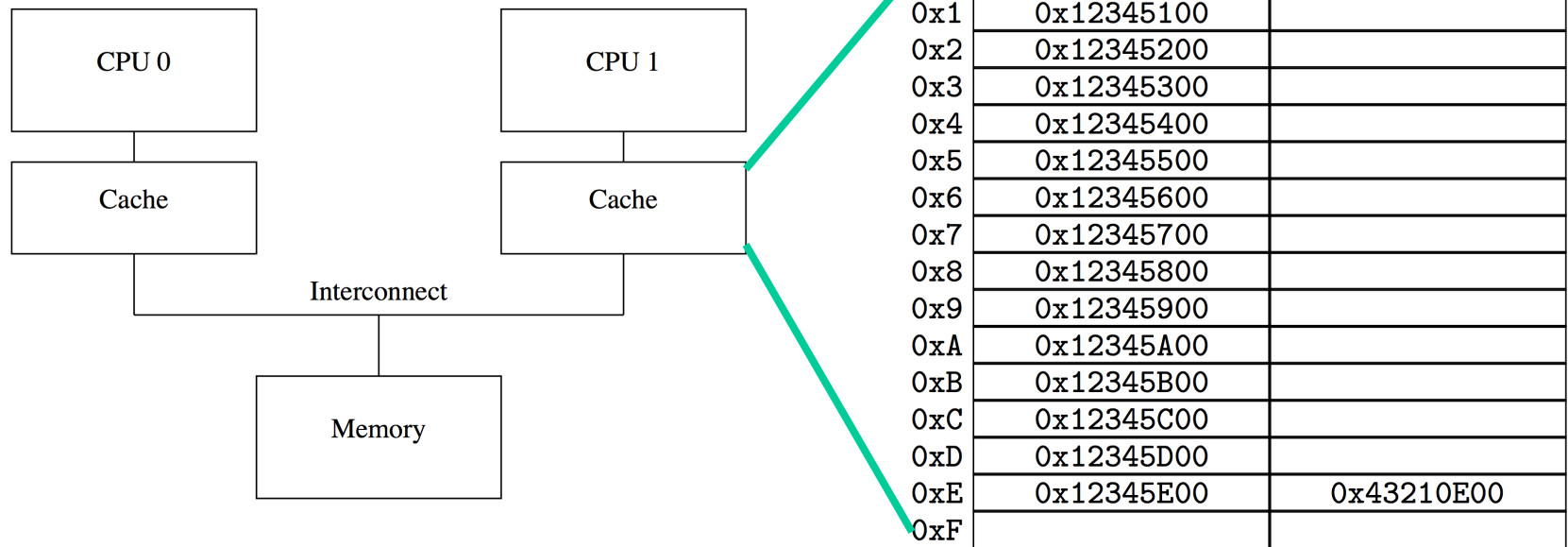
A typical multicore CPU with three levels of cache



- Floating-point register add or mul: 0.4 ns
- RAM access: > 80 ns (single-thread read)

Fix 1: Each processor core has a cache

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



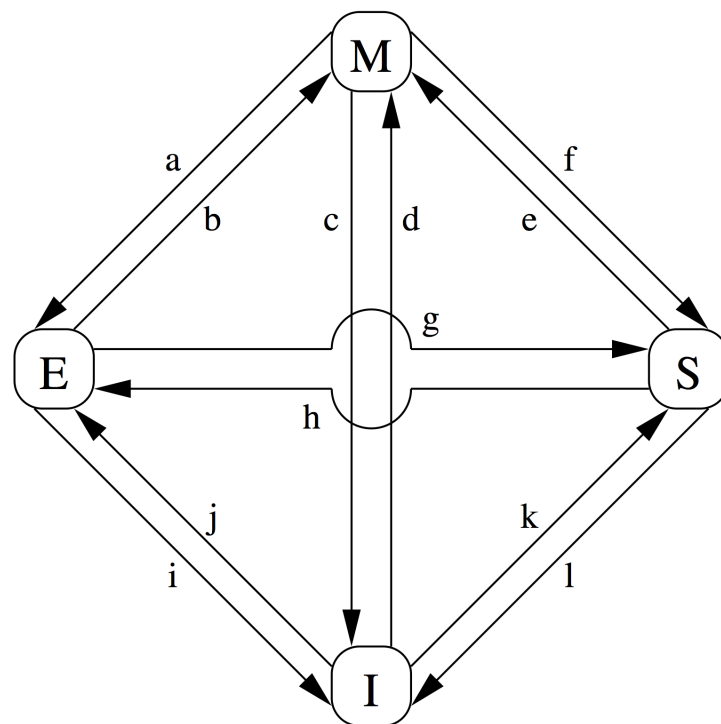
- Two caches may have different values for a given memory address

Fix 2: Get all caches to agree

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

State	Cache line	Excl	RAM	To read	To write
M odified	Modified by me	Y	stale	from cache	to cache
E xclusive	Not modified	Y	fresh	from cache	to cache -> M
S hared	Others have it too	N	fresh	from cache	send invalidate
I nvalid	Not in use by me	-	-	elsewhere	send invalidate

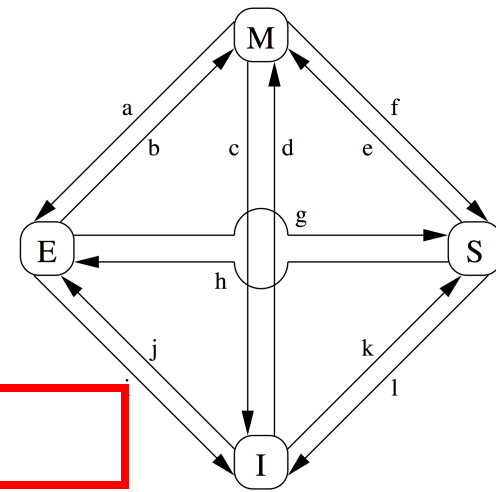
- A cache line
 - has 4 possible states
 - and 12 transitions a-l
- Cache messages
 - sent by cores to others
 - via cache bus
 - to make caches agree



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



		Cause	I send	I receive	My response
M	a	(Send update to RAM)	writeback	-	-
E	b	Write	-	-	-
M	c	Other wants to write	-	read inv	read resp, inv ack
I	d	Atomic read-mod-write	read inv	read resp, inv ack*	-
S	e	Atomic read-mod-write	read inv	inv ack*	-
M	f	Other wants to read	-	read	read resp
E	g	Other wants to read	-	read	read resp
S	h	Will soon write	inv	inv ack*	-
E	i	Other wants atomic rw	-	read inv	read resp, inv ack
I	j	Want to write	read inv	read resp, inv ack*	-
I	k	Want to read	read	read resp	-
S	l	Other wants to write	-	inv	inv ack 12

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 cache line “owned” by another core

			This core wants to	Messages	Speed
Unshared mutable	M	M	Read cache line	0	fast
	M	M	Write cache line	0	fast
	E	E	Read cache line	0	fast
Shared immutable	E	M	Write cache line	0	fast
	S	S	Read cache line	0	fast
Shared mutable	I	S	Read cache line	1+1	slow
	S	M	Write cache line	1+N	very slow
	I	M	Write cache line	1+1+N	very slow

N cores

One more performance problem: “false sharing” because of cache lines

- A cache line typically is 64 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 7 longs!
- Frequently written memory locations should not be on the same cache line!
 - even if apparently not shared between threads
- 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

```
class NewLongAdderPadded {
    private final static int NSTRIPES = 31;
    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();
        }
    }
}
```

Avoid side-by-side AtomicLong allocation

- Allocate many AtomicLongs
 - instead of AtomicLongArray

Unless JVM is too clever

- Scatter the AtomicLongs
 - by allocating some Objects in between

Don't do this at home

Plan for today

- Atomic longs with "striping"
- Memory, cache and cache coherence
 - Shared mutable data on multicore is slow
- **Graphical user interface toolkits, 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

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

Lock Ordering

- Two threads: 1,2
- Two locks: A,B
- Thread 1: acquire first lock A, then lock B
- Thread 2: acquire first lock B, then lock A
- If both manage to acquire 'their' first lock, both get stuck: **DEADLOCK**

- If there is a global total order of all locks and all threads acquire locks in this order, the above cannot happen.

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/kggh/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

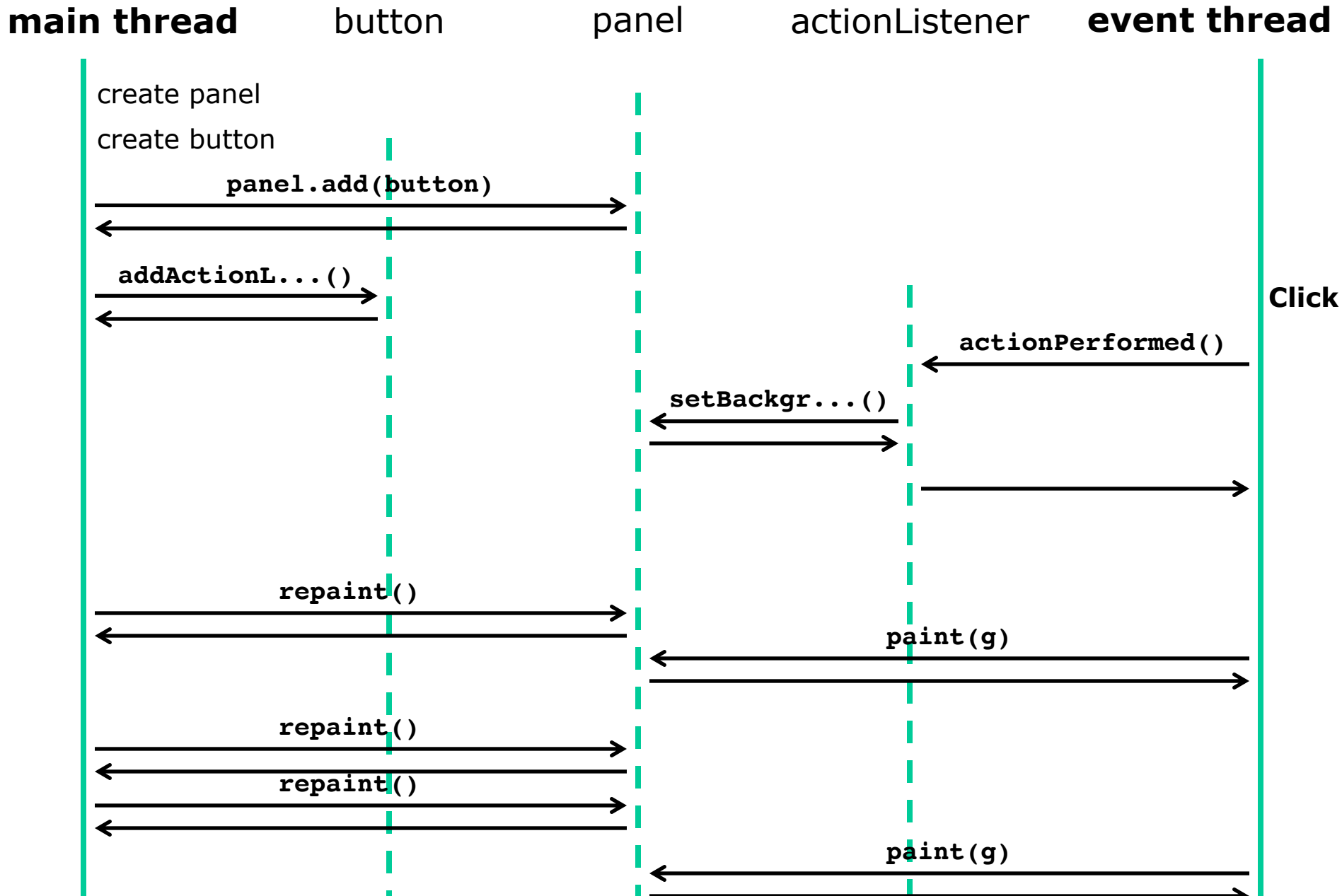
A short computation on the event thread

```
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);
```

TestButtonGui.java

- 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

Main thread and event thread



Using the main thread for blinking

```
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 **paint(g)** later

Fetching webpages on event thread

```
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()));  
        }  
    }  
});
```

On event
thread

TestFetchWebGui.java

Bad

- 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

Fetching web with SwingWorker

```
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();
    }
}
```

On worker thread

Computed result

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

Get result

On event thread

TestFetchWebGui.java

- `SwingWorker<T,V>` implements `Future<T>`
- .NET has similar `System.ComponentModel.BackgroundWorker`

Fetching web with SwingWorker

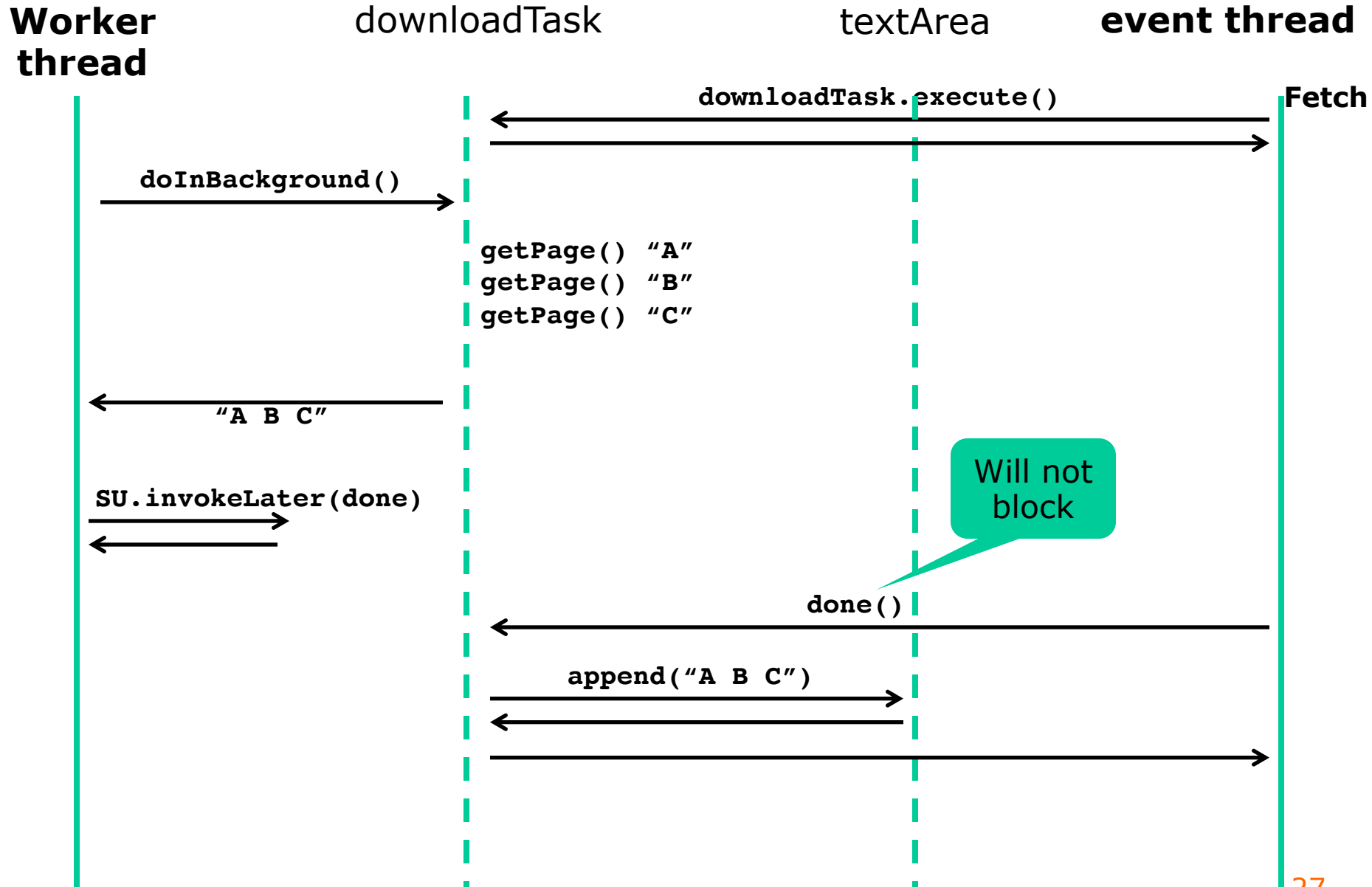
```
DownloadWorker downloadTask = new DownloadWorker(textArea);
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.execute();
    }
});
```

- 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

Dogma 1

Dogma 2

Worker thread and event thread



Add progress notification

```
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();  
    }  
}
```

On worker
thread

- In the GUI setup, add:

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

On event
thread

Add cancellation

```
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"); }  
    } }  
}
```

On worker
thread

- In the GUI setup, add:

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

On event
thread

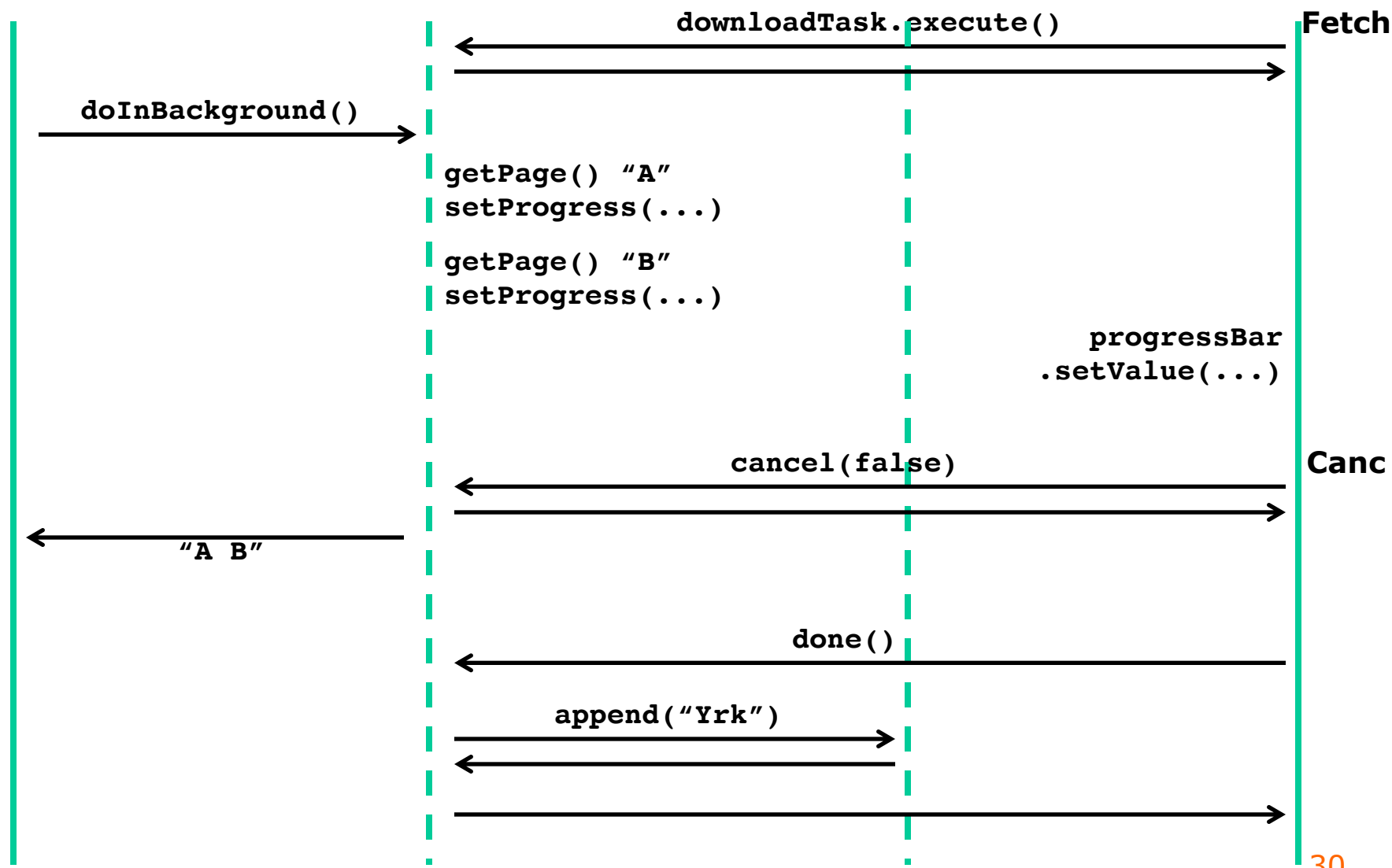
Progress and cancellation

Worker thread

downloadTask

textArea

event thread



Show results gradually

```
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);  
        }  
    }  
}
```

On worker
thread

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

On event
thread

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

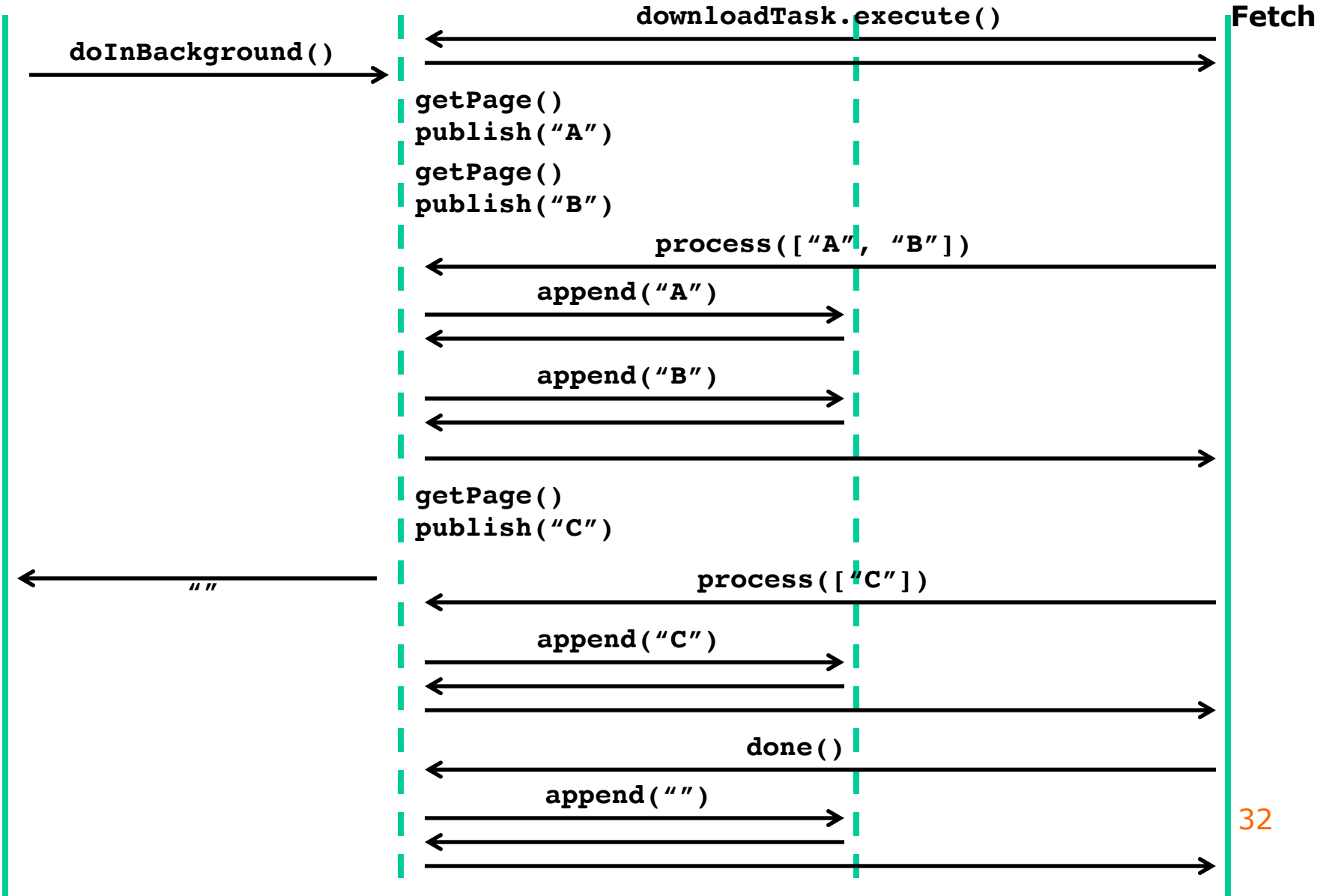
Event thread and downloadTask

Worker thread

downloadTask

textArea

event thread



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()**

Very proper GUI creation in Swing

as per <http://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html>

```
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);
        }
    });
}
```

TestButtonGuiProper.java

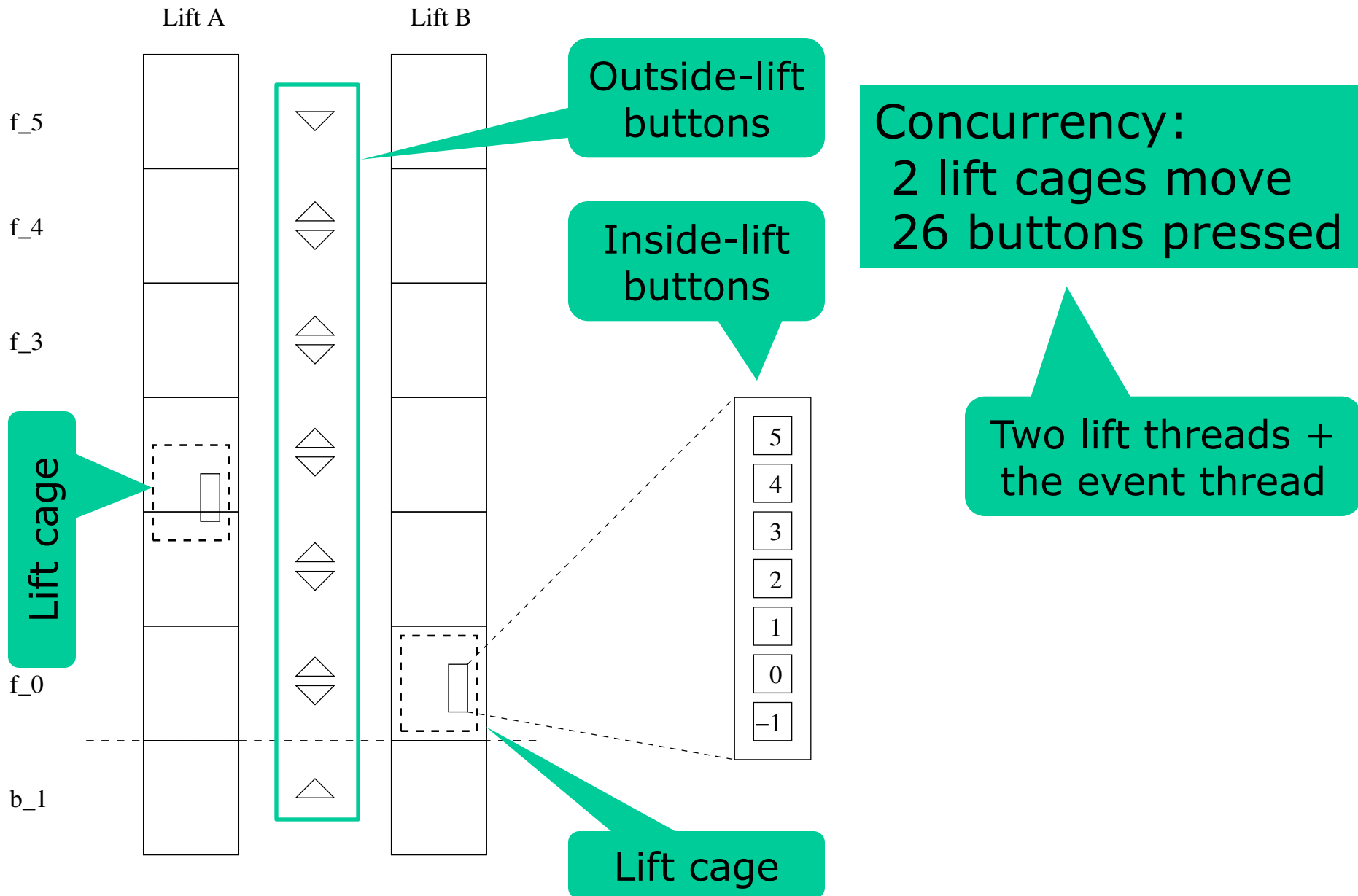
GUI gets built on the Event Thread

- Avoids interaction with a partially constructed GUI
 - because the Event Thread is busy constructing the GUI

Plan for today

- Atomic long with “thread striping”
- Shared mutable data on multicore is slow
- 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**

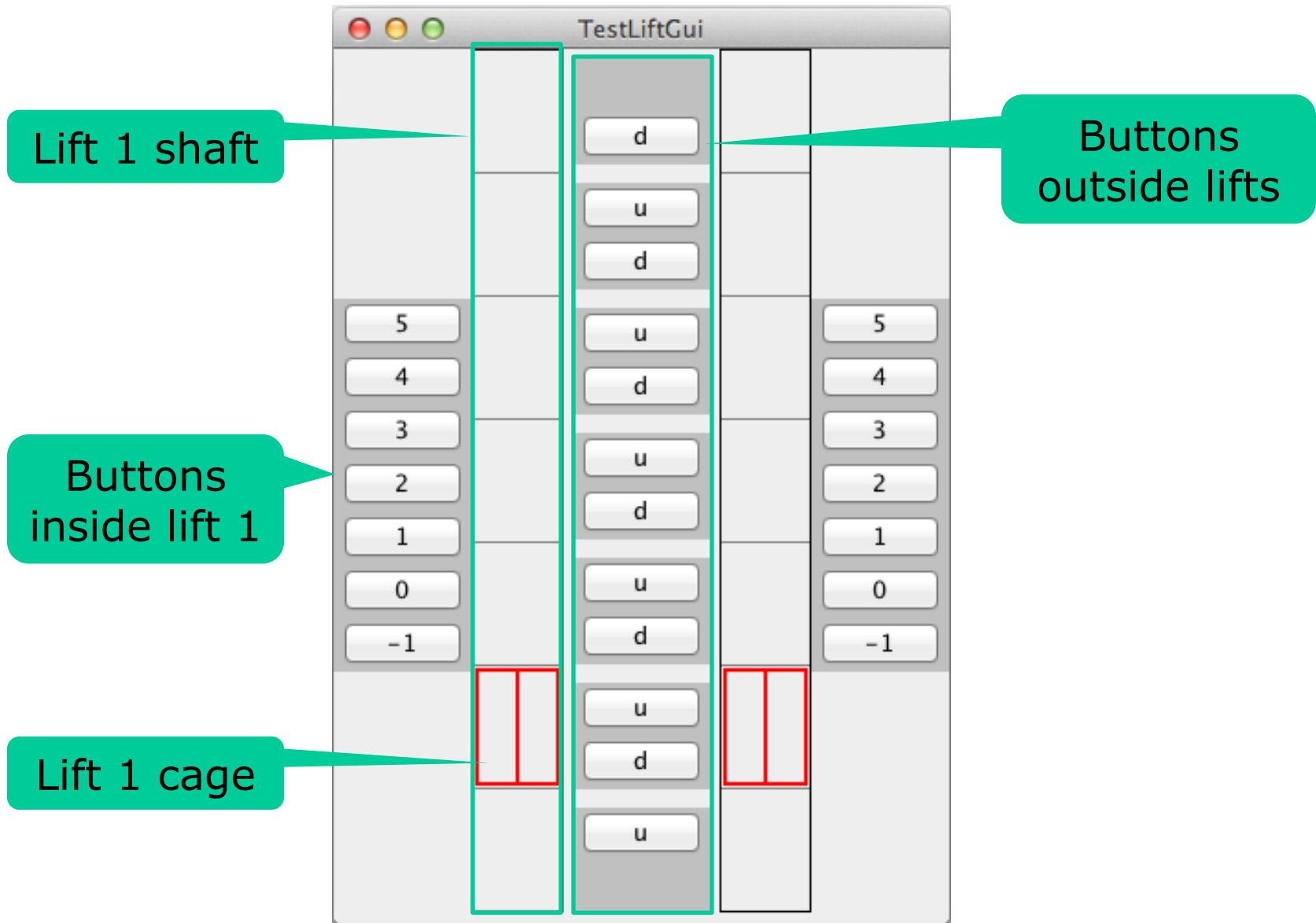
Example: 2 lifts, 7 floors, 26 buttons



Modeling and visualizing the lifts

- Use event thread for button clicks (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 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

```
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);
    }
}
```

Up or Down

Ask lifts[i]
how long

Choose the
soonest one

The state of a lift

- Current floor and direction (None, Up, Down)
- required stops and directions, **stops[n]**:
 - **null**: no need to stop at floor **n**
 - **None**: stop at floor **n**, don't know future direction
 - **Down**: stop at floor **n**, then continue down
 - **Up**: stop at floor **n**, then continue up
 - **Both**: stop, then up, and later down; or vice versa

```
class Lift implements Runnable {
    private double floor;
    private Direction direction; // None, Up, 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

Accessed on lift and
controller threads

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

```
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: ... dual to Up ...
case None: ... if any stops[floor] != null, start moving in that direction ...
}
```

Executed 16 times/second

on lift thread

TestLiftGui.java

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 later calls `paint(g)`, OK
- Lift and event threads access `stops[]` array
 - guarded by lock on lift instance `this`
- Only lift thread accesses `floor` and `direction`
 - not guarded by a lock

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, timeouts?
E.g. if the lift sent to floor 4 going Up gets stuck at floor 3 by some fool blocking the open door?
- 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 12-13

Armstrong et al: Concurrent Programming in Erlang (1993) 11.1

This week

- Reading this week
 - Goetz et al chapter 9, 10.1, 10.2
 - McKenney: *Memory barriers*, chapters 1-4
- Exercises
 - You can write responsive and correct user interfaces involving concurrency
- Read before next week's lecture
 - Goetz chapter 12
 - Herlihy and Shavit chapter 3