

CS 351

Design of Large Programs

Object Design: Safety and Liveness

October 26, 2021

Software Assurance: Testing

Testing is an integral part of software development

- is a necessity at all levels
- is expensive
- can be simplified through the use of tools
- illuminates the existence of bugs
- fails to establish the absence of bugs
- is assisted by debugging tools and skills

Software Assurance: Testing

Concurrency makes testing exceedingly difficult

- Program A
 - 10 boolean variables
 - 2^{10} possible states
- Program B
 - 10 boolean variables
 - 2^{10} possible states
- Concurrent execution
 - $2^{10} \times 2^{10}$ possible states
 - $2^{10} = 1024$
 - $2^{20} = 1048576$

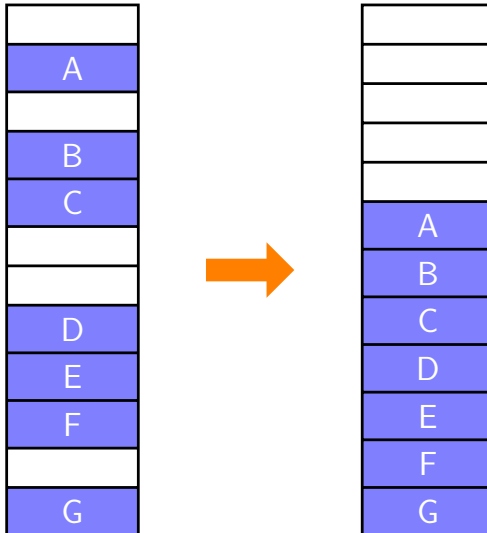
Software Assurance: Verification

- Formal verification
 - provides strong guarantees
 - is difficult to carry out
 - requires specialized knowledge and tools
- Good design facilitates both testing and verification

Key Concepts in Verification

- Safety – the program does nothing wrong
 - data integrity
 - absence of deadlock
- Liveness – the program does something
 - reaches a desired goal
 - terminates

Example: Memory Compaction



Specification

- Safety
 - the contents of each used memory block remains unchanged
 - the ordering of used memory blocks remains unaltered
 - the number of free space blocks remains unchanged
- Liveness
 - the protocol reaches a state in which all free space is contiguous located at the top

Safety Properties

- *Invariant* – a property that holds initially and forever
 - block A holds data XX
 - block A is above block C
- *Stable* – a property which, once established, holds forever
 - all blocks below k^{th} block are used blocks
 - all blocks above k^{th} block are free blocks

Liveness Properties

- *Leads to* – from a state S1 the Memory Compaction program eventually reaches a state S2
 - S1: any memory state
 - S2: all free space is at the top
- *Metric* – progress is measured by a decrease in some useful metric
 - sum of distances from the top of the memory to each free block
 - initial value: $0 + 2 + 5 + 6 + 10 = 23$
 - final value: $0 + 1 + 2 + 3 + 4 = 10$

Important Observations

- The protocol may be implemented as a sequential or concurrent program
- Concurrent threads should not interfere with each other
 - this is an important additional proof obligation
 - the proof may be highly complex
 - the program may be vulnerable to subtle errors
- It is desirable to ensure *correctness by design*
 - at least when it comes to non-interference (atomicity)

Design of Safe Objects

Conservative design strategies

- Immutability – avoiding state changes
- Synchronization – ensuring mutual exclusion
- Containment – structural restrictions ensuring exclusive access

Immutable Objects: Reliance on Constructors

- Fixed variables can be initialized in the constructor
- No methods to change the variable are provided

```
public class Leader {  
    private final NodeId n;  
  
    public Leader(NodeId n) {  
        this.n = n;  
    }  
    // no setter for field!  
}
```

Immutable Objects: Stateless Methods

- Methods that have no bearing on the state of the object
- Pure functions
 - return value depends only on the passed arguments
 - return value may also depend upon immutable variables

```
public class NumericalOps {  
    public static int add(int a,  
                           int b) {  
        return a + b;  
    }  
}
```

```
public class Leader {  
    private final NodeId n;  
    //...  
    public int rank(NodeId k) {  
        if (k.id() < n.id()) {  
            return n.id() - k.id();  
        } else {  
            return 0;  
        }  
    }  
}
```

Immutable Objects: Copying Policy

- Making a local temporary copy and returning it as the result of the computation protects the original object
- Copy needs to be atomic in order to avoid destroying data integrity
- The object passed as argument needs to offer an atomic copy method

```
public int[] sort(int[] array) {  
    int[] copy = new int[array.length];  
    // place sorted elements in copy...  
    return copy;  
}
```

Synchronized Objects: Full Synchronization

- The goal is to ensure mutual exclusion among all the methods accessing the object
 - every method is synchronized
 - no public fields are present
- Access to other object can break encapsulation!
- Careful analysis and discipline is required

Display Room Lighting

- Assume that we have a display room with multiple light sources
- one and only one light is on at any one time
- a user can select a light to turn on
- a user can turn off the light, forcing some other light to be turned on

Synchronized Object: Display Room Lighting

```
public class LightControl {
    private List<Light> lights;
    private Light onLight;
    private Random rand;
    // initialize fields...

    public synchronized void on(Light light) {
        if (onLight != null) onLight.turnOff();
        onLight = light;
        onLight.turnOn();
    }

    public synchronized void off() {
        if (onLight != null) onLight.turnOff();
        onLight = lights.get(rand.nextInt(lights.size()));
        onLight.turnOn();
    }
}
```

Static Field Complications

- Synchronization assures mutual exclusion among methods of the same object
- Methods of different objects can interfere with each other if they access static fields
- Options:
 - allow only static synchronized methods to access static fields
 - use block synchronization with a lock on that class
 - getClass()

Protecting Static Fields: File Users Counter

```
public class FileUsers {  
    private static int userCount = 0; // never access directly  
  
    protected void beginUsing() {  
        synchronized (getClass()) {  
            ++userCount;  
        }  
    }  
  
    protected void endUsing() {  
        synchronized (getClass()) {  
            --userCount;  
        }  
    }  
  
    public static synchronized int numberUsing() {  
        return userCount;  
    }  
}
```

Partial Synchronization

- Only methods that can interfere with each other are declared as synchronized
- Only sections of code where interference can occur are protected by a synchronization block

```
synchronized(this) {  
    // code here...  
}
```

Partial Synchronization: Linked List

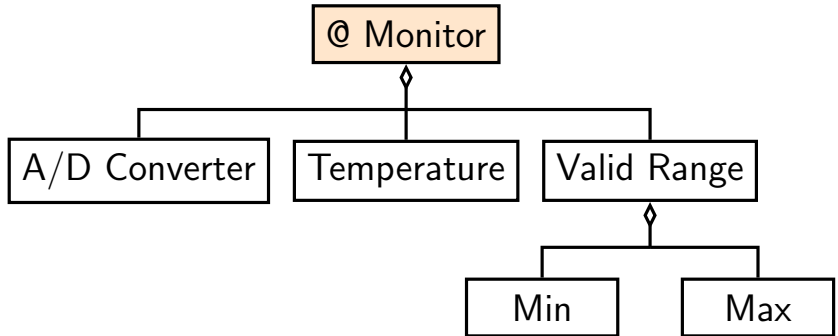
```
public class LinkedList {  
    protected double value;  
    protected final LinkedList next;  
  
    public LinkedList (double value, LinkedList next) {  
        this.value = value;  
        this.next = next;  
    }  
  
    public synchronized double getValue() { return value; }  
  
    public synchronized void setValue(double value) {  
        this.value = value;  
    }  
  
    public LinkedList getNext() { return next; }  
  
    public double sum() { // add up element values  
        double v = getValue(); // get value via synchronized accessor  
        if (next() != null) { v += next().sum(); }  
        return v;  
    }  
}
```

Contained Objects: Exclusive Ownership

The outer object

- ensures that only one thread executes at a time (synchronized methods)
- subordinate objects are locally created
- references to the subordinate objects are not leaked
 - not passed as arguments
 - not passed as returned values

Exclusive Ownership Example



Contained Objects: Managed Ownership

- It is often the case that ownership of a resource may change over time
- Invariant: *only one accessible reference exists in the system at any one time*
- Ownership transfer operations must be subject to defined policies enforced by design

Liveness: Failure Modes

- Reliance on timing properties
 - proofs of concurrent programs are meant to show that under all possible interleaving of events the execution is correct
- Contention on the CPU leading to starvation
 - proofs of concurrent programs assume that no thread is denied service by the OS
- Dormancy – a suspended thread never becomes schedulable again
 - suspend/resume and wait/notify pairing errors
- Premature termination
- Deadlock

Deadlock Prevention

- Every thread locks the resources it needs in the same order
- The locking order is not always readily visible
- Complications arise when locks are deep inside the object nesting structure

```
public class Document {  
    private Document otherPart;  
    public synchronized void print() {  
        // print this part of the document  
    }  
    public synchronized void printAll() {  
        otherPart.print();  
        print();  
    }  
}
```

Deadlock Illustration

Thread 1

Thread 2

Deadlock Illustration

Thread 1

letter.printAll

letter is locked

Thread 2

enclosure.printAll

enclosure is locked

Deadlock Illustration

Thread 1

letter.printAll

letter is locked

letter.otherPart.print

blocked

waiting for enclosure

Thread 2

enclosure.printAll

enclosure is locked

enclosure.otherPart.print

blocked

waiting for letter

Additional Failure Modes

- Data integrity violations
 - synchronized methods do not always guarantee atomicity
 - data inconsistencies lead to incorrect decisions
- Lack of progress
 - progress made by one thread is undone by another
- Livelock
 - threads take turns yielding to each other

Livelock Illustration

Thread 1

resource requested

Thread 2

resource requested

Livelock Illustration

Thread 1

resource requested
contention on
resource?

Thread 2

resource requested
contention on
resource?

Livelock Illustration

Thread 1

resource requested
contention on
resource?

Yes!

Thread 2

resource requested
contention on
resource?

Yes!

Livelock Illustration

Thread 1

resource requested
contention on
resource?

Yes!

withdraw request

Thread 2

resource requested
contention on
resource?

Yes!

withdraw request

Livelock Illustration

Thread 1

resource requested
contention on
resource?

Yes!

withdraw request

try later

Thread 2

resource requested
contention on
resource?

Yes!

withdraw request

try later

Livelock Illustration

Thread 1

resource requested
contention on
resource?

Yes!

withdraw request

try later

Thread 2

resource requested
contention on
resource?

Yes!

withdraw request

try later

How could we stop colliding?

Locking and Performance

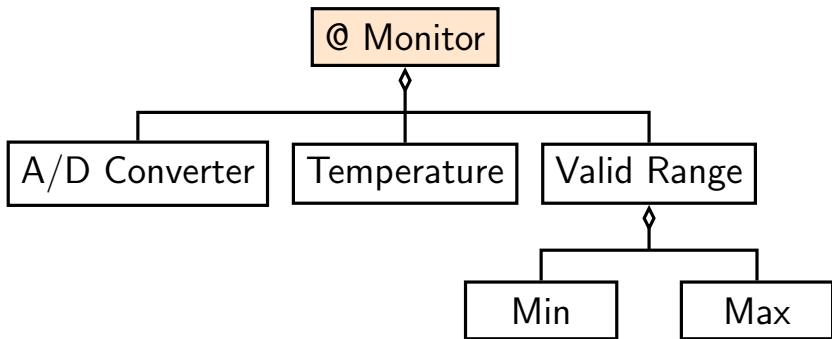
- Synchronized methods and locks may introduce performance penalties
 - correctness must come first
 - thoughtful analysis can ensure correctness and enhance performance
- Instant variable analysis helps determine where synchronization is or is not needed
- Proper design of object hierarchies may simplify analysis and reduce the need for synchronization

Interleaving Semantics

- Access and update methods are normally synchronized
- A very long update can delay reading of the data
- Two solutions:
 - employ synchronization blocks
 - let the access method be unsynchronized as long as it returns one of only two values:
 - the value *before* the update
 - the value *after* the update

A Conservative Design Revisited

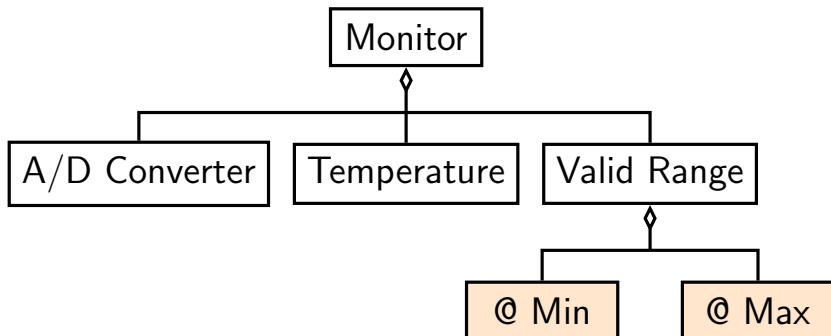
- All methods of Monitor are synchronized
 - protection against any data inconsistencies
- Only one thread can use Monitor at a time



Class Splitting

- Monitor can be redesigned
 - by using synchronization only on contained objects that need it
- Multiple threads can use Monitor concurrently
- More generally, contained objects may be designed in order to achieve fine-grained synchronization and increased levels of concurrency

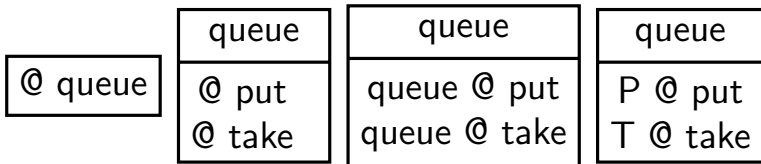
Class Splitting



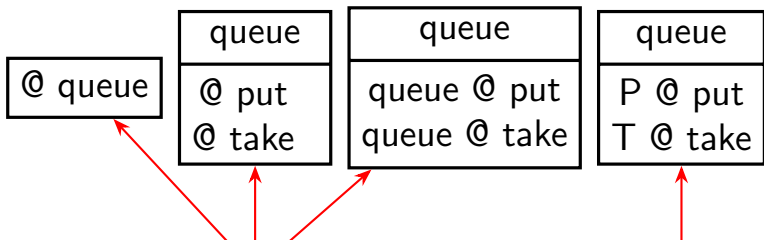
Lock Splitting

- Every Java Object can be a lock
- Synchronized methods use the object to which they belong as a lock
- Another object could be used to accomplish the same objective
- Using multiple locks allows subgroups of methods to be mutually exclusive
- Going one step further, locks can be selectively used in a state dependent manner

Sample Design Notation



Sample Design Notation



All methods synchronized on queue

Separate locks
for put and take