The goal of this lab is to understand how to add a new algorithm to Synchrobench Java. Refer to Lab 1 for the setup of the Java version of Synchrobench.

Exercise 1: Data Structure Implementing a Type

Synchrobench offers numerous data structures, several synchronization techniques, and few abstract data types. The difference between structures and types is that the abstract data type specifies an interface while the data structure is a possible implementation.

For example, the list-based set is simply a set abstract data type implemented using a linked list data structure. Sometimes the distinction is less clear: a queue is an abstract data type that specifies the behaviour of enqueue/dequeue but it can be implemented with different data structures, an array and a sorted linked list, etc.

The Set abstraction does not accept duplicated elements and generally exports simple methods:

- `add(v)` that returns false if \( v \) is absent from the set, otherwise it inserts value \( v \) to the set and returns true;
- `remove(v)` that returns false if \( v \) is not in the set, otherwise it deletes value \( v \) from the set and returns true;
- `contains(v)` that returns true if value \( v \) is present is the set or false if \( v \) is absent from the set.

There exist other abstract data types, like a collection or a multi-set that allows duplicates and a map or a dictionary that maps a key to a value, however, the set is one of the simplest abstract data types and we will illustrate how to implement a set using a linked list that uses a single global lock as the synchronization technique.

Duration: 5 min

Exercise 2: Reflection, Interfaces and Abstract Classes

Synchrobench is a benchmark suite and each of its benchmark represents a data structure implementing a type with a given synchronization technique. Hence, Synchrobench offers, for example, multiple list-based set benchmarks. While they all implement a set using a sorted linked list data structure, they use different synchronization techniques like compare-and-swap (CAS), transactional memory (TM) or locks, and sometimes these benchmarks even use different ways to synchronize with locks: with classic locks or with versioned try-locks, etc.
As Synchrobench uses reflection to load a new benchmark specified with `-b`, a new class has to implement some interface (`CompositionalSortedSet`, `CompositionalMap`) or to extend the abstract class (`AbstractCompositionalIntSet`) depending on the type to implement. These interfaces and abstract classes are located in `contention/abstractions/`. Hence a benchmark (e.g., `linkedlists.lockbased.VersionedListSet`) can be dynamically selected by the `contention.benchmark.Test` class at runtime as indicated on the command line:

```
java -cp bin contention.benchmark.Test -b linkedlists.lockbased.VersionedListSet
```

**Exercise 3: Single Global Lock for List-Based Set**

For simplicity, here is an example of the implementation of a list-based set using a global lock for synchronization. This benchmark is not intended to be efficient, but is used for illustration here.

Synchrobench allows to write new algorithm that implements a set abstraction of element of a generic type and also integer set abstraction (containing elements only of type int). For simplicity, we show how to implement an integer set abstraction below.

Extending the abstract class `AbstractCompositionalIntSet` requires to implement the two methods (`size()` and `clear()`). Note that these are just placeholders (for simplicity). Now, to make sure Synchrobench will print the actual size at the end of the execution (useful for sanity checks) it is important to implement a sequential (not necessarily thread-safe) size() method.

To use Synchrobench with a non-null JVM warmup and multiple iterations within the same JVM instance, then it is also important to make sure that `clear()` empty the linked list. The clear method is called between the warmup and the actual benchmark to empty the structure. If the cleanup is not implemented and a non-null warmup is specified as a parameter (or the default warmup is used), then the benchmark may not terminate.

```java
public class MyCoarseGrainedListIntSet extends AbstractCompositionalIntSet {
    final private Node head;
    final private Node tail;
    private Lock lock = new ReentrantLock();

    public MyCoarseGrainedListIntSet() {
        head = new Node(Integer.MIN_VALUE);
        tail = new Node(Integer.MAX_VALUE);
        head.next = tail;
    }

    public boolean addInt(int item) {
        lock.lock();
        try {
            Node pred=head;
            Node curr=head.next;
            while (curr.key < item) {
                pred = curr;
                curr = pred.next;
                ...
if (curr.key==item) {
    return false;
} else {
    Node node = new Node(item);
    node.next=curr;
    pred.next=node;
    return true;
}
} finally {
    lock.unlock();
}

public boolean removeInt(int item) {
    lock.lock();
    try {
        Node pred=head;
        Node curr=head.next;
        while (curr.key < item) {
            pred = curr;
            curr = pred.next;
        }
        if (curr.key==item) {
            pred.next=curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        lock.unlock();
    }
}

public boolean containsInt(int item) {
    lock.lock();
    try {
        Node pred=head;
        Node curr=head.next;
        while (curr.key < item) {
            pred = curr;
            curr = pred.next;
        }
        return (curr.key == item);
    } finally {
        lock.unlock();
    }
}

public int size() {

```java
int size = 0;
Node pred;
Node curr = head.next;
while (curr.next != null) {
    pred = curr;
    curr = pred.next;
    size++;
}
return size;
}

public void clear () {
    head.next = tail;
}
```

Note that the class MyCoarseGrainedListIntSet extends AbstractCompositionalIntSet and is located in package linkedlists.lockbased. Once this new class gets added to Synchrobench in the directory src/linkedlists/lockbased, one can simply compile the code and run it using:

```
1 cd synchrobench/java
2 make
3 java -cp bin contention.benchmark.Test -W 0 -b linkedlists.lockbased.MyCoarseGrainedListIntSet
```

Again make sure to implement proper clear() and size() methods when testing with a non-null warmup.

*Duration: 20 min*

**Exercise 4: Next Steps**

Take a look at “The Art of Multiprocessor Programming” or the recent literature to add new Java benchmark to Synchrobench.