Replication and Consistency

07 The Universality of Consensus

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Thank you!

These slides are based on companion material of the following books:

- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

Previously on Replication and Consistency

- Consensus number characterizes synchronization power of objects
- There is no wait-free implementation of X by Y

Previously on Replication and Consistency

- Consensus number characterizes synchronization power of objects
- There is no wait-free implementation of X by Y
- Can we implement objects with consensus number 1, 2, ... from a class of objects that has consensus number ∞?

Universality

A class C is **universal** if one can construct a wait-free implementation of *any object* (in some "universe") from arbitrarily many objects of C and arbitrarily many read-write registers.

Universality

A class C is **universal** if one can construct a wait-free implementation of *any object* (in some "universe") from arbitrarily many objects of C and arbitrarily many read-write registers.

From n-process consensus, we can construct a

- wait-free
- linearizable
- n-threaded implementation
- of **any** sequentially specified object!

Theorem(Herlihy 1991)

A class is universal in a system of n threads if and only if it has consensus number $\geq n$.

Proof Outline

- We will show a universal construction
  - From n-consensus objects
  - And atomic registers
- Not a practical construction
- But we know where to start looking!

Generic Sequential Objects

- **Initial state**
- Apply sequence of method **invocations**
  - method name + parameters
- Each invocation has a **response**
  - termination condition (normal / exceptional)
  - return value (if any)
- Here: Deterministic objects!

```java
public interface SeqObject {
  public abstract Response apply(Invocation invoc);
}

public class Invoc {
  public String method;
  public Object[] args;
}

public class Response {
  public Object value;
}
```

- Similar to Bakery algorithm

Example: Stack

- Initial state: empty
- For the following invocation sequence, what are the return values for the invocations?

```
push(2)  -- push(4) -- pop() -- push(3) -- push(2) -- pop()
```

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Universal Construction: Idea

- Object represented as
  - initial object state
  - a log, i.e. a linked list of the method calls
- For new method call:
  - Find head of list
  - Atomically append call
  - Compute response by traversing the log while applying all invocations (upto and including the new one) on a private copy
  - Return result

Linearizing concurrent invocations

- Use *one-time* consensus object to decide next entry in log
- All threads update the corresponding pointer based on decision from consensus
  - OK because they all write the same value
- Tail of log is immutable, only updates at head of the log
  - Allows concurrent executions of `apply(...)` on private copy
- Unsuccessful thereads need to run yet another consensus on the new head

Linearizing concurrent invocations

- Use *one-time* consensus object to decide next entry in log
- All threads update the corresponding pointer based on decision from consensus
  - OK because they all write the same value
- Tail of log is immutable, only updates at head of the log
  - Allows concurrent executions of `apply(...)` on private copy
- Unsuccessful thereads need to run yet another consensus on the new head

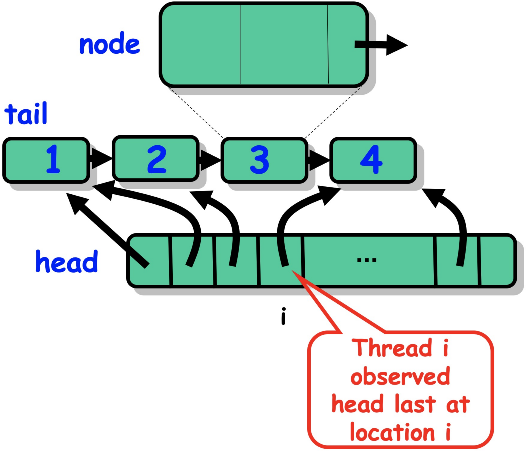
What happens if a thread stops some point while executing these steps?

Representation of Log Entries

```
class Node {
  Invoc invoc;
  Consensus<Node> decideNext;
  Node next;
  int seq;   // sequence number

  Node(Invoc invoc) {
    this.invoc = invoc;
    this.decideNext = new Consensus<Node>()
    this.seq = 0;  // 0 indicates that node is not in log yet
  }
```

Universal Object



Remarks

- Consensus objects only work **once**
  - Trick: Each node has its own consensus object
- Maximum value in `heads` array is current actual head of log
  - Similar to Bakery algorithm

Universal Object

```
class Universal {
  Node[] head;
  Node tail = new Node();
  tail.seq = 1; // sentinel node

  Universal() {
    for (int j = 0; j < n; j++){
      head[j] = tail;
    }
  }

  static Node max(Node[] array) {
    Node max = array[0];
    for (int i = 1; i < array.length; i++)
      if (max.seq < array[i].seq)
        max = array[i];
    return max;
  }
...
```

Universal Application - Part 1

```
Response apply(Invoc invoc) {
  int i = ThreadID.get();
  // construct new log entry object
  Node prefer = new Node(invoc);

  // while not added to the list
  while (prefer.seq == 0) {
    // node at head of list where I will try to append
    Node before = Node.max(head);
    // run consensus proposing my new node
    Node after = before.decideNext.decide(prefer);
    // set next pointer based on position; potentially done by
    multiple threads
    before.next = after;
    // set sequence number, indicating that node has been inserted
    after.seq = before.seq + 1;
    // update my knowledge of log list
    head[i] = after;
    }
   // to be continued
```

Universal Application - Part 2

```
...
  // initial version of my private copy of object
  SeqObject MyObject = new SeqObject();
  // iterate over log and apply all invocations up to my own one
  current = tail.next;
  while (current != prefer){
    MyObject.apply(current.invoc);
    current = current.next;
  }
  // return response for my own current invocation
  return MyObject.apply(current.invoc);
}
```

Correctness of Construction

- List defines linearized sequential history
  - Linearization point when consensus is decided for a node
- Thread returns its response based on list order

Correctness of Construction

- List defines linearized sequential history
  - Linearization point when consensus is decided for a node
- Thread returns its response based on list order

Is the construction wait-free?

Correctness of Construction

- List defines linearized sequential history
  - Linearization point when consensus is decided for a node
- Thread returns its response based on list order

Is the construction wait-free?

- Append at head is done in finite number of steps
- But: Threads can be fail repeatedly when trying to win the consensus
- However, this implies other threads make progess!

Progress conditions

Lock-freedom

In an infinite execution, infinitely often **some** method call finishes (obviously, in a finite number of steps).

Wait-freedom

Each method call takes a finite number of steps to finish.

Correctness: Lock-freedom

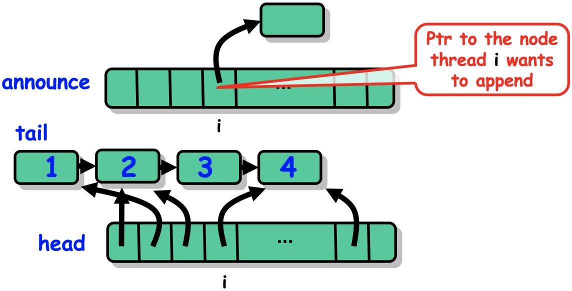
Our universal construction so far is *lock-free* because:

- Thread can repeatedly fail to win consensus on head only if another succeeds
- Consensus winner adds node and completes within a finite number of steps

From lock-free to wait-free

- Idea: Threads **help** each other to append their nodes
- Need to make additional information available for supporting threads
- Will reuse lock-free construction with additional **announce** array
  - Store (pointer to) node in announce
  - If a thread doesn't append its node, another thread will see it in the array and help to append it

Wait-free Universal Object



Adding the Announce Array

```java
public class Universal {
  private Node[] announce; // additional array with n entries
  private Node[] head;
  private Node tail = new node();

  Universal() {
    tail.seq = 1;
    for (int j = 0; j < n; j++){
      head[j] = tail;
      announce[j] = tail;   // initiallly set to sentinel node
    }
  }
```

A Cry for Help!

```
public Response apply(Invoc invoc) {
  int i = ThreadID.get();
  // announce my new log entry
  announce[i] = new Node(invoc);
  // find head of list
  head[i] = Node.max(head);
  while (announce[i].seq == 0) {
  ...
  // while node not appended to list
  ...
  }
```

Zooming into the loop

```
while (announce[i].seq == 0) {
  Node before = head[i];
  Node help = announce[(before.seq + 1) % n];
  if (help.seq == 0)
     prefer = help;
  else
     prefer = announce[i];
...
```

- Non-zero sequence number indicates success
- Thread keeps helping append nodes until its own node is appended

Help!

- When last node in list has sequence number k, all threads check whether thread $(k+1) \mod n$ wants help
  - If so, try to append its node first
- In general, after (max) n more nodes appended, some thread will have observe that thread $k+1$ wants help
  - Try to append that node
  - Some threads succeeds
- After thread i announces its node, no more than n other calls can start and finish without appending i's node

Finishing the Job

- Once a thread's node is inserted in the log, the rest is again the same as in lock-free algorithm
- That is: Compute the result by sequentially applying the method calls in the list to a private copy of the object starting from the initial state

QED

Implications

Theorem(Herlihy 1991)

A class is universal in a system of n threads if and only if it has consensus number $\geq n$.

- `getAndSet()` is not universal for system with $\geq n$ threads
- `compareAndSwap()` is universal for any number of threads

Any architecture that does not provide a universal primitive has inherent limitations!

- You cannot avoid locking for concurrent data structures!
- But why do we care? Is locking really so bad?

Locking and Scheduling

- Locking affects the assumptions we need to make on the operating system in order to guarantee progress
- The **scheduler** is a part of the OS that determines
  - which thread gets to run on which processor
  - how long it runs for
- A given thread can be **active**, that is, executing instructions, or **suspended**.

Do You Remember these Progress Conditions?

??: Some thread trying to acquire the locks eventually succeeds.

??: Every thread trying to acquire the locks eventually succeeds.

??: Some thread calling the method eventually returns.

??: Every thread calling the method eventually returns.

Solution: Progress conditions

Deadlock-free: Some thread trying to acquire the locks eventually succeeds.

Starvation-free: Every thread trying to acquire the locks eventually succeeds.

Lock-free: Some thread calling the method eventually returns.

Wait-free: Every thread calling the method eventually returns.

Schedulers are usually benevolent

- Programmers design lock-free or deadlock-free algorithms, but what they are implicitly assuming is that all method calls eventually complete as if they were wait-free.
- Schedulers are do not single individual threads out, but are fair.

Next on Replication and Consistency

- We learned how to define the safety (correctness) and liveness (progress) of concurrent programs and objects
- We are ready to start the practice of implementing them
- Next lecture: Implementing spin locks on multiprocesor machines!

Further reading

Herlihy, Maurice. 1991. "Wait-Free Synchronization." *ACM Trans. Program. Lang. Syst.* 13 (1): 124–49. https://doi.org/10.1145/114005.102808.