Simulating Read / Write Objects

Can we provide a shared read / write variable in an asynchronous message-passing system, when processes can fail?
- Yes, if we have enough nonfaulty processes

Can we provide stronger types of read / write variables, when processes can fail?
- Yes, as long as we don’t read-and-write
Abstract Data Types (ADT)

- Cover many concurrent applications
- Abstract representation of data & operations for accessing it
  - Signature
  - Specification

Implementing High-Level ADT

Using lower-level ADTs & procedures

High-level operations translate into primitives on base objects
- read, write
- compare&swap
- read-modify-write

Low-level (primitive) operations are often implemented from more primitive operations
- A hierarchy of implementations
Interleaving Operations’ Executions

Invocation

P₁

P₂

P₃

Response
Sequential executions

Sequential execution: invocations & responses alternate and match (on process & object)

Sequential specification: All legal sequential executions, satisfying ADT’s semantics
  – E.g., stack: pop returns the last item pushed

Correctness: Linearizability

- For every concurrent execution, there is a sequential execution of the same operations that
  - Is legal (obeys the specification of the ADTs), and
  - Preserves the real-time order of non-overlapping operations
- Equivalently, each operation appears to take effect instantaneously at some point between its invocation and its response (atomicity)
- When processes fail (there is a partitioning of processes into faulty and nonfaulty), this holds for all completed operations and a subset of the pending operations

Some operations never complete
Linearizability is Composable (Local)

- The whole system is linearizable ⇔ each object is linearizable
- Allows to implement and verify objects separately

Example: Simulating Shared Memory

- Provide a single-writer single-reader register (this is the high-level) in a message-passing system
  - Accessed by read and write operations
- Underlying system is asynchronous message passing (this is the low-level), where less than half the processes can crash
Linearizability

Simulating Shared Memory w/ Failures

- Requires a majority of nonfaulty processes
- Otherwise, the system can be partitioned
  - A read “misses” the latest write
Must have $n > 2f$

**Theorem:** A simulation of a 1-reader, 1-writer read/write linearizable register in an asynchronous message passing tolerates at most $f < n/2$ crash failures

**Proof:** Suppose in contradiction there is an algorithm tolerating $f = n/2$ crash failures

Partition processes into two sets, $Q_0$ and $Q_1$, each of size $f$

---

Must have $n > 2f$: Writing

Consider an execution in which

- initial value of simulated register is 0
- all processes in $Q_1$ crash initially
- process $p_0$ in $Q_0$ invokes write(1) at time 0 and no other operations are invoked
- the write completes at some time $t_0$ without any process in $Q_0$ receiving a message from any process in $Q_1$
Must have n > 2f: Reading

Consider another execution in which

– initial value of simulated register is 0
– all processes in Q₀ crash initially
– process p₁ in Q₁ invokes a read at time t₀+1 and no other operations are invoked
– the read completes at some time t₁ without any process in Q₁ receiving a message from any process in Q₀
– the read returns 0, due to linearizability

Must have n > 2f: Arithmetic

Now paste the views of processes in Q₀ from the first execution with the views of processes in Q₁ from the second execution

– messages between Q₀ and Q₁ are delayed to arrive after time t₁

This execution is not linearizable, since read(0) follows write(1)

→ Must assume a majority of nonfaulty processes
The Algorithm in a Nutshell: Write

- The simulated register is replicated at each process
- Each data item has a unique sequence number
  - sequence of values
- \texttt{write(d, val, seq#)}
  - generate next sequence number
  - send a message with the value and the sequence number to all processes
  - each recipient updates its replica and sends ack
  - writer waits for n-f > n/2 acks

The Algorithm in Action: Write
The Algorithm in Action: Write

value: 1
value: 1
value: 0

write ok
data

write ok

write 1

The Algorithm in a Nutshell: Read

• Each data item has a unique sequence number

• read(d) returns (val, seq#)
  – send a request to all processes
  – each recipient sends back current value of its replica
  – wait for > n/2 replies
  – return value associated with largest sequence number
  – do a write-back to ensure atomicity of reads
The Algorithm in Action: Read

Key Idea for Correctness

- Each read should return the value of "the most recent" write
- Each read or write communicates with > n/2 processes
  ➔ The set of processes communicating with a read intersects the set of processes communicating with a write

- Since system is asynchronous, a message on behalf of an operation might be overtaken by a message on behalf of a later operation
  – reader and writer keep track of "status" of each link
  – don't send a message on a link before receiving ack on previous message (ping-pong)
Proving Linearizability

Let $ts(W) =$ sequence number of $W$
Let $ts(R) =$ sequence number of write that $R$ reads from
$O_1 \rightarrow O_2$ denotes $O_1$ completes before $O_2$ starts

Key lemmas:
• If $W_1 \rightarrow W_2$, then $ts(W_1) < ts(W_2)$
• If $W \rightarrow R$, then $ts(W) \leq ts(R)$
• If $R \rightarrow W$, then $ts(R) < ts(W)$
• If $R_1 \rightarrow R_2$, then $ts(R_1) \leq ts(R_2)$

Simulating R/W Registers from R/W Registers

- single-reader single-writer binary-valued
- single-reader single-writer multi-valued
- multi-reader single-writer multi-valued
- multi-reader multi-writer multi-valued
- atomic snapshots
Atomic Snapshots

- \( m \) components
- **Update** a single component
- **Scan** all the components “at once” (atomically)

Provides an instantaneous view of the whole memory, very useful for designing shared-memory algorithms

Has a wait-free implementation from read/write variables

---

Atomic Snapshots: More Formally

Operations are
- invocation \( \text{scan}_i \), returns \( V \), where \( V \) is an array of \( n \) values
- invocation \( \text{update}(i,d) \) where \( d \) is a data value, returns \( \text{ok} \)

Legal sequences: if \( V \) is returned by a scan, \( V[k] \) is the parameter of latest preceding \( \text{update}(k,*) \)

For example:

\[
\text{update}(1,x) \quad \text{update}(2,y) \quad \text{scan}([a,x,y]) \quad \text{update}(0,z) \quad \text{scan}([z,x,y])
\]
Atomic Snapshots: Key Ideas

• Store each component in a separate variable
• To update: write to the respective variable
• To scan: Collect (read) values of the segments twice
  – If no segment is updated during the "double collect",
    this is a valid snapshot -- return it
• How to tell if a segment is updated?
  – Tag each value with a sequence number (1,2,3,...)

Atomic Snapshots: Partial Algorithm

\[
\begin{align*}
\text{Update}(k,v) & \quad \text{Scan()} \\
A[k] = \langle v, \text{seq}_i, i \rangle & \quad \text{repeat} \\
\end{align*}
\]

Linearize:
• Updates with their writes
• Scans inside the double collects
**Atomic Snapshot: Linearizability**

Double collect (read a set of values twice)
If equal, there is no write between the collects
– Assuming each write has a new value (seq#)

\[ \text{read } A[1],\ldots,A[m] \quad \text{read } A[1],\ldots,A[m] \]

\[ \text{write } A[j] \]

Creates a “safe zone”, where the scan can be linearized

---

**Wait-free Atomic Snapshot**

Embed a scan within the Update, and write its view to the segment
Scanner returns view obtained in last collect

\[ \text{Update} (v, k) \]
\[ V = \text{scan} \]
\[ A[k] = \langle v, \text{seq}_i, V \rangle \]

**Linearize:**
- Updates with their writes
- Direct scans as before
- Borrowed scans with source
Atomic Snapshot: Borrowed Scans

Interference by process $p_j$
And another one...
$\Rightarrow p_j$ does a scan inbetween

\[
\begin{array}{cccc}
\cdots & \cdots & \cdots & \cdots \\
\text{write } A[j] & \text{embedded scan} & \text{write } A[j] \\
\end{array}
\]

Linearizing with the borrowed scan is OK.

Complexity of Atomic Snapshots

- Uses $O(m)$ read/write variables (some are large)
- Scan needs $O(n^2)$ reads and writes, why?
- Update needs $O(n^2)$ reads and writes
Simulating R/W Registers from R/W Registers

multi-reader multi-writer multi-valued

atomic snapshots

multi-reader single-writer multi-valued

single-reader single-writer binary-valued

Multi-Valued From Binary

The simulated register takes values \( \{0, \ldots, K-1\} \)

**Binary** approach: a different binary register stores each bit of the multi-valued register being simulated

- Read algorithm reads all registers and returns the resulting value
- Write algorithm writes the new bits in some order

Errors when the reader overlaps a slow write and sees some new bits and some old bits
A Unary Approach

Use an array of $K$ binary registers, $B[0..K-1]$
- value $v$ is represented with $B[v] = 1$ and other entries 0

- **Read** algorithm: read $B[0]$, $B[1]$, ..., until finding the first 1; return the index
- **Write** algorithm: set new entry of $B$ and zero the old entry of $B$

OK if reads and writes don't overlap.

When Reads and Writes Overlap...

**Problem:** reader may never find a 1 in $B$
**Solution:** write algorithm only clears (sets to 0) entries that are smaller than the entry that is set (to 1)
New-old inversion

Corrected Algorithm

**Read:** scans up to first 1, *then read down* to check those entries are still 0; return smallest index set during downward read

**Write(r):** set r to 1 and then set to 0 entries smaller than r

Clearly, **wait-free:**
- writer does at most $K$ (primitive) writes
- reader does at most $2K-1$ (primitive) reads
Linearization Proof for Multi-Valued Construction

Fix an admissible execution of the algorithm

– Primitive operations (binary read / write) are atomic

We give a permutation of the (high-level) operations that is legal (by construction)

Show it respects real-time ordering of non-overlapping operations

Reads-From Relations

Primitive read r of a binary register B[v] reads from primitive write w to B[v] if w is the latest write to B[v] that precedes r in the execution

High-level read R reads from high-level write W if R returns v and W contains the primitive write that R's last primitive read of B[v] reads from
The Permutation

Primitive read $r$ of a binary register $B[v]$ reads from primitive write $w$ to $B[v]$ if $w$ is the latest write to $B[v]$ that precedes $r$ in the execution.

High-level read $R$ reads from high-level write $W$ if $R$ returns $v$ and $W$ contains the primitive write that $R$'s last primitive read of $B[v]$ reads from.

- Place (high-level) writes in the order they occur — no concurrent writes.
- Consider each (high-level) read $R$ in the order they occur — no concurrent reads.
- If $R$ reads from write $W$, place $R$ immediately before the write that follows $W$ in the permutation.

Legal by construction.

Permutation Preserves Real-Time Order

- **write-write**: OK, by construction.
- **read-write**: OK, since cannot read from a later write.
- Two cases remain:
  - write-read
  - read-read

**Lemma**: If high-level read $R$ returns $v$, and $R$ read of any $B[u]$, $u < v$, during its upward scan, reads from a (primitive) write contained in high-level write $W$. Then $R$ does not read from a write that precedes $W$. 
Lemma: If high-level read $R$ returns $v$, and $R$ read of any $B[u], u < v$, during its upward scan, reads from a (primitive) write contained in high-level write $W$. Then $R$ reads from a write that follows $W$. 

Proof of Lemma

By contradiction, if $w > u$, write 1 to $B[w]$, write 0 to $B[u]$, write $v$ to $B[v]$, read $v$ from $B[v]$, read 0 from $B[u]$, read 1 from $B[v]$, during upward scan, $u < v$ or during downward scan.

\[ w > u \]

If $w > v$?

Lemma: If high-level read $R$ returns $v$, and $R$ read of any $B[u], u < v$, during its upward scan, reads from a (primitive) write contained in high-level write $W$. Then $R$ reads from a write that follows $W$. 

Proof of Lemma

By contradiction, if $w > u$, write 1 to $B[w]$, write 0 to $B[u]$, write $v$ to $B[v]$, read $v$ from $B[v]$, read 0 from $B[u]$, read 1 from $B[v]$, during upward scan, $u < v$ or during downward scan.

\[ w > u \]
Simulating R/W Registers from R/W Registers

- single-reader single-writer binary-valued
- single-reader single-writer multi-valued
- atomic snapshots
- multi-reader single-writer multi-valued
- multi-reader multi-writer multi-valued

Multi-Reader from Single-Reader: 1st Attempt

Use n single-reader registers

write: write new value in each Val[i] register
read: return value from own Val[i] register

- Val[0] = 0
- Val[n] = 0

write 1
read 1
read 0
new-old inversion

- P_w
- write 1 to Val[1]
- read 1 from Val[1]
- write 1 to Val[2]
- P_1
- read 0 from Val[1]
- P_2
- read 0 from Val[2]
Readers Must Write

**Theorem:** In a wait-free simulation of a multi-reader single-writer register from single-reader single-writer registers, at least one reader writes

**Proof:** Suppose, in contradiction, there is an algorithm in which readers never write

- \( p_w \) is the writer, \( p_1 \) and \( p_2 \) are the readers
- initial value of simulated register is 0
- \( S_1 \) are the single-reader registers read by \( p_1 \)
- \( S_2 \) are the single-reader registers read by \( p_2 \)

Readers Must Write

- Consider execution in which \( p_w \) writes 1 to the simulated register, by a series of writes, \( w_1, \ldots, w_k \), to the single-reader registers.
  - Each of them is either in \( S_1 \) or in \( S_2 \) (but not both)

\[ \begin{align*}
p_w & \quad \text{write } w_1 \quad \ldots \quad \text{write } w_j \quad \text{write } w_{j+1} \ldots \quad \text{write } w_k
\end{align*} \]
Readers Must Write

$v_j^i$ denotes the value returned if $p_i$ reads after $w_j$

For each reader $p_i$ the value of the simulated register "switches" from 0 (old) to 1 (new), at some point

- $v_1^1 = \ldots = v_{a-1}^1 = 0$, $v_a^1 = \ldots = v_k^1 = 1$
- $w_a$ is a write to a register in $S_1$
- $v_1^2 = v_2^2 = \ldots = v_{b-1}^2 = 0$, $v_b^2 = \ldots = v_k^2 = 1$
- $w_b$ is a write to a register in $S_2$

Assume $a < b$

Since readers do not write, they return the same values

$\Rightarrow$ old-new inversion, not linearizable
Corrected Multi-Reader Algorithm

In the simulated read, announce the value to be returned

Check values returned by previous reads

Sequence numbers allow to compare returned values

Writer's Algorithm

• get the next sequence#  
  – an integer, incremented by 1 each time
• write (value, sequence#) to Val[1],...,Val[n]  
  (one copy for each reader)
Reader $p_i$'s Algorithm

- read (value, sequence#) from Val[$i$]
- read (value, sequence#) from Report[$j,i$]
- pick pair with largest sequence#
- write that pair to row $i$ of Report
- return value component of that pair

Correctness of Multi-Reader Algorithm

- Obviously wait-free
  - Write: $n$ primitive writes
  - Read: $n+1$ primitive reads and $n$ primitive writes
- To prove linearizability, show a permutation of the high-level operations that is clearly legal and then prove it preserves real-time order of non-overlapping operations.
Constructing the Permutation

• Put in all writes in the order they occur in the execution
  – Single writer \( \Rightarrow \) writes do not overlap
• Consider the reads in the order of their responses in the execution
  – read \( R \) reads from write \( W \) if \( W \) generates the sequence\# associated with the value \( R \) returns
  – place \( R \) immediately before the write that follows \( W \)
• By construction, the permutation is legal.

Preserving Real-Time Order

• \textbf{write-write}: by construction
• \textbf{read-write}: \( R \) precedes \( W \) in the execution. Then \( R \) cannot read from \( W \) or any later write.
  \( \Rightarrow \) \( R \) is placed before \( W \) in the permutation
• \textbf{write-read}: \( W \) precedes \( R \) in the execution. Then \( R \) reads \( W \)'s sequence\# or a larger one from \( \text{Val}[ ] \) and reads from \( W \) or a later write.
  \( \Rightarrow \) \( R \) is placed after \( W \) in the permutation.
• \textbf{read-read}: \( R_i \) by \( p_i \) precedes \( R_j \) by \( p_j \) in the execution. Then \( p_j \) reads \( R_j \)'s sequence\# or a larger one from \( \text{Report}[i,j] \). So \( R_j \) reads from the same write that \( R_j \) reads from or a later write
  \( \Rightarrow \) \( R_j \) is placed after \( R_i \) in the permutation.
Simulating R/W Registers from R/W Registers

- single-reader
  - single-writer
  - binary-valued
- single-reader
  - single-writer
  - multi-valued
- multi-reader
  - single-writer
  - multi-valued
- atomic
  - snapshots
- multi-reader
  - multi-writer
  - multi-valued

Multi-Writer from Single-Writer: Key Ideas

- Each writer announces each value it wants to write to all the readers, by writing the value to its own (single-writer multi-reader) register
- Each reader reads all the values written by the writers and returns the latest one
- How to determine latest value?
  - use timestamps (as in Bakery algorithm)
  - since multiple processes generate timestamps, need to coordinate timestamp generation
Multi-Writer from Single-Writer

✓ Wait-free by construction
Createlinearization:
- Place writes in timestamp order
- Insert each read before the write following the write it returns

Add logical time to values

Write\( (v,X) \)
read \( T S_1,\ldots, read \ T S_n \)
\( T S_i = \max T S_j + 1 \)
write \( (v,T S_i,i) \) to \( R_i \)

Read\( (X) \)
read \( R_1,\ldots, read \ R_n \)
return \( v_j \) with maximal \( T S_j,j > \)

Multi-Writer from Single-Writer

✓ Wait-free by construction
Createlinearization:
- Place writes in timestamp order
- Insert each read before the write following the write it returns

✓ Legality is immediate
✓ Real-time order is preserved since a read returns a value (with timestamp) larger than all preceding operations
Beyond Registers

Registers support *read* and *write* operations
Wait-free simulations of one kind of register out of another kind (numbers of values, readers, writers)

What about (wait-free) simulating a significantly different kind of data type out of registers?
More generally, what about (wait-free) simulating an object of type X out of objects of type Y?

---

Key Insight

• Focus on asynchronous, wait-free simulations
  – Typically, in shared memory

Ability to simulate object of type X using only objects of type Y and registers is related to the ability of those data types to solve consensus!
Example: FIFO Queue

- **invocation** `enq(x)` and response `ack`
- **invocation** `deq` and response `ret(x)`

Each `deq` returns oldest enqueued value that is not yet been dequeued (`⊥` if queue is empty)

**Can we simulate a queue with registers?**
No! Otherwise, can solve 2-processes consensus w/ registers:
- simulate a FIFO queue using registers
- use consensus algorithm with queues and simulated queue to solve 2-process consensus

**Can we simulate CAS with queues?**
Not with > 3 processes! Otherwise, can solve 3-processes consensus w/ queues:
- simulate CAS using queues (and registers)
- use CAS-based consensus and simulated CAS to solve 3-processes consensus
Another Example: Multi-Assignment

• Atomic snapshot means
  – Write any array element
  – Read multiple array elements atomically
• What about multi-assignment
  – Write multiple array elements atomically
  – Scan a subset of array elements

**Theorem:** There is no wait-free implementation of multi-assignment from read / write registers

◊ Compare with atomic snapshots

---

Multi-Assignment: Proof Strategy

• Solve 2-process consensus
• Take a 3-element array
  – A writes atomically to slots 0 and 1
  – B writes atomically to slots 1 and 2
  – Any thread can scan any set of locations
• Impossible with atomic registers ⇒

**Theorem:** There is no wait-free implementation of multi-assignment from read / write registers
Consensus with Multi-Assignment

Uses one multi-assignment object A with 3 entries, all initially EMPTY

Code for process $i = 0$ or $i = 1$

```plaintext
propose[i] = input
A.assign2(i, i, i+1, i)
other = A.read((i+2) mod 3)
if (other == EMPTY or other == A.read(1))
    return propose[i];
else return propose[1-i];
```

Consensus with Multi-Assignment

$p_0$ wins if $p_1$ didn’t move or $p_1$ moved later

```plaintext
propose[i] = input
A.assign2(i, i, i+1, i)
other = A.read((i+2) mod 3)
if (other == EMPTY or other == A.read(1))
    return propose[i];
else return propose[1-i];
```
Consensus with Multi-Assignment

\[ \text{p}_0 \text{ loses if } \text{p}_1 \text{ moved earlier} \]

\[
\begin{align*}
\text{propose}[i] &= \text{input} \\
A.\text{assign2}(i, i, i+1, i) \\
\text{other} &= A.\text{read}((i+2) \mod 3) \\
\text{if} \ (\text{other} == \text{EMPTY} \text{ or } \text{other} == A.\text{read}(1)) \\
\text{return } \text{propose}[i]; \\
\text{else return } \text{propose}[1-i];
\end{align*}
\]

Consensus Numbers

Data type \( X \) has consensus number \( \text{CN}(X) = n \) if \( n \) is the largest number of processes for which consensus can be solved using only objects of type \( X \) and read/write registers.

Determine if there is a wait-free simulation of \( Y \) from \( X \) based on their consensus number.
Consensus Numbers

Data type $X$ has **consensus number** $CN(X) = n$ if $n$ is the largest number of processes for which consensus can be solved using only objects of type $X$ and read/write registers

<table>
<thead>
<tr>
<th>data type</th>
<th>consensus number</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write register, snapshots</td>
<td>1</td>
</tr>
<tr>
<td>FIFO queue, fetch&amp;Inc</td>
<td>2</td>
</tr>
<tr>
<td>compare &amp; swap</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

**Theorem:** If $CN(X) > CN(Y) = m$, then there is no wait-free simulation of an object of type $Y$ using objects of type $X$ and read/write registers for $> m$ processes

Consensus Numbers: Proof

Assume there is a wait-free simulation of $Y$ using $X$ and registers in a system with $k$, $n \geq k > m$ processes

Consensus algorithm for $k$ processes
- Since $CN(Y) = n$, there is a $k$-process consensus algorithm using $Y$ and registers
- Execute this algorithm using simulated objects of type $Y$ from objects of type $X$ (and registers)

**Theorem:** If $n = CN(Y) > CN(X) = m$, then there is no wait-free simulation of an object of type $Y$ using objects of type $X$ and read/write registers for $> m$ processes
Sample Corollaries

There is no wait-free simulation of any object with consensus number > 1 using read/write registers.

There is no wait-free simulation of any object with consensus number > 2 using queues and read/write registers.

There is no wait-free simulation of any object with consensus number > 2 using 2-assignment objects and read/write registers.

Universality of Consensus Numbers

Data type is universal if objects of that type and read/write registers can wait-free simulate any data type.

**Theorem:** A data type with consensus number \( n \) is universal in a system with \( \leq n \) processes.

1. A non-blocking \( n \)-process algorithm to simulate any data type using **compare & swap**
2. Modify to use objects with consensus number \( n \)
3. Modify to be wait-free
4. Bound the shared memory used and handle non-determinism
Universal Simulation Using CAS

Represent object by a linked list with the sequence of operations applied to the simulated object.

Apply an operation on the simulated object by inserting an appropriate node at the head of the linked list.

Use compare&swap on the Head pointer of the list.

The Linked List

Each linked list node has
- operation invocation (= type and parameters)
- new state of the simulated object
- operation response
- pointer to previous node (= previous op)
**Simulation w/ CAS: In Pictures**

- Head
  - invocation
  - state
  - response
  - before
  - initial state
  - anchor

- point
  - invocation
  - state
  - response
  - before

If CAS indicates that Head changed, try again with the new Head.

**Simulation w/ CAS: The Code**

Initially Head points to anchor node
- represents initial state of simulated object

Local variables h, point

When inv is invoked:

```
allocate a new linked list node in shared memory, pointed to by local var point
point.inv = inv
repeat
  h = Head
  point.state, point.response = apply(inv, h.state)
  point.before = h
until compare&swap(Head, h, point) == h
```

- do the output indicated by point.response
- depends on simulated data type
- Head not changed, point it to new node

© Hagit Attiya 2013: R/W simulations
Strengthening the Algorithm 1: Using Arbitrary Consensus Objects

Replace compare&swap object with an arbitrary n-process consensus object

Use it to decide on the next operation

Since a consensus object is one-shot, use many copies, each allowing to thread an additional operation to the list.

Strengthening the Algorithm 1: Finding the Head of the List

Per-process Head pointer, to the last node it has inserted

Sequence numbers allow to identify the latest node.
Algorithm with Consensus Objects

Initially all Head entries point to the anchor node

when inv occurs
    point = new opr, point.inv = inv
for j=0 to n-1
    find node with maximum sequence number
    if Head[j].seq > Head[i].seq then Head[i]=Head[j]
repeat
    try to thread your operation
    win = decide(Head[i].after, point)
    win.seq = Head[i].seq+1
    win.new-state, win.response =
        apply(win.inv, Head[i].new-state)
    Head[i]=win
    point to the following node
until win = point
return point.response

Non-Blocking Universal Simulation: In Pictures

Successful case

- 0
- 1
- i
- n-1
- Head
- point
- seq
- seq
- 0
- initial state
- anchor
Strengthening the Algorithm 2: Making it Wait-Free

- Use **helping**: processes help each other to finish pending operations (not just their own)
- When appending the \( r^{th} \) operation, help process \( j = r \mod n \) (if \( j \) has a pending operation)
- **Announce** array stores pending operations

Announce

0 1 j n-1

0
invocation
--
--
after

Code for Wait-Free Simulation

Initially all Head and Announce entries point to anchor

When inv occurs

\[
\text{Announce}[i] = \text{new opr, Announce}[i].\text{inv,seq} = \text{inv,0}
\]

\[
\text{for } j = 0 \text{ to } n - 1
\]

\[
\text{if Head}[j].\text{seq} > \text{Head}[i].\text{seq then Head}[i] = \text{Head}[j]
\]

\[
\text{while Announce}[i].\text{seq} == 0 \text{ do}
\]

\[
\text{priority} = \text{Head}[i].\text{seq} + 1 \mod n
\]

\[
\text{if Announce[priority].seq} == 0 \text{ then}
\]

\[
\text{point} = \text{Announce[priority]}
\]

\[
\text{else point} = \text{Announce}[i]
\]

\[
\text{win} = \text{decide(Head}[i].\text{after}, \text{point)}
\]

\[
\text{win.new-state, reponse} = \text{apply(win.inv, Head}[i].\text{new-state)}
\]

\[
\text{win.seq} = \text{Head}[i].\text{seq} + 1
\]

\[
\text{Head}[i] = \text{win}
\]

\[
\text{return Announce}[i].\text{response}
\]
Strengthening the Algorithm 3: Bounding the List

- A process allocates nodes from a private pool
- A node is recycled when it is not referenced anymore

When can we recycle node #r?

- No process trying to thread node \((r+n+1)\) or higher will access node \(r\)
- When the operations that thread nodes \(r\ldots r+n\) terminate, node \(r\) can be recycled
- When process \(p\) finishes threading node \(m\) it releases nodes \(m-1\ldots m-n\). After node \(r\) is released by the operations threading nodes \(r\ldots r+n\), it can be recycled