# Wait-Free Synchronization
# Lecture 1

*CS380D—Distributed Computing*
*The University of Texas at Austin*

# Coöperation

- *Many large problems require multiple processes to cooperate on a solution*
- *Cooperation requires shared data*

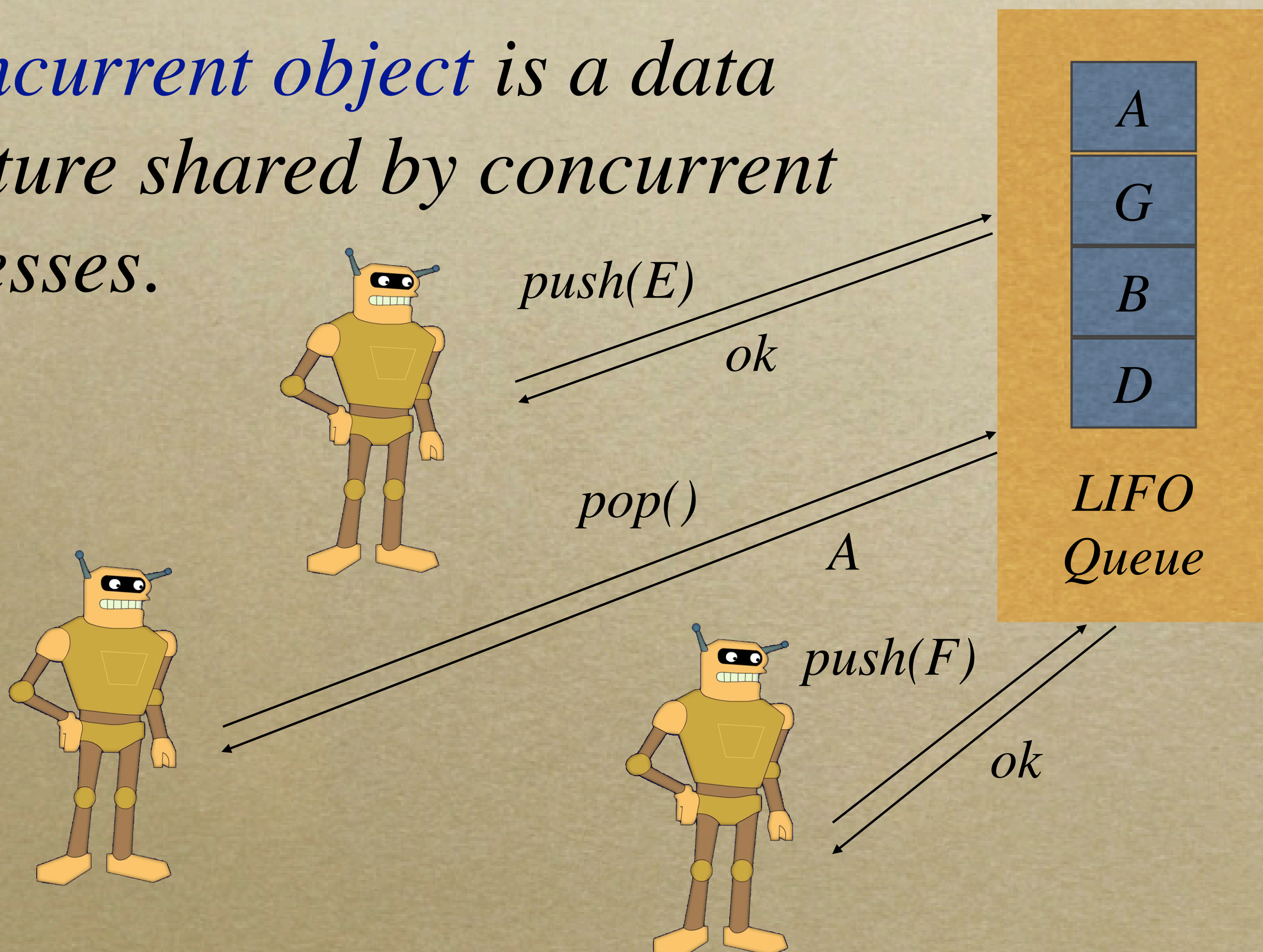
How can we provide efficient concurrent access to shared data?

# Concurrency Model (Informal)

- *Asynchronous—concurrent processes execute without relative bounds on the speed between processes, and a process's speed may vary over time.*
- *Failstop—processes may fail by halting at any time.*

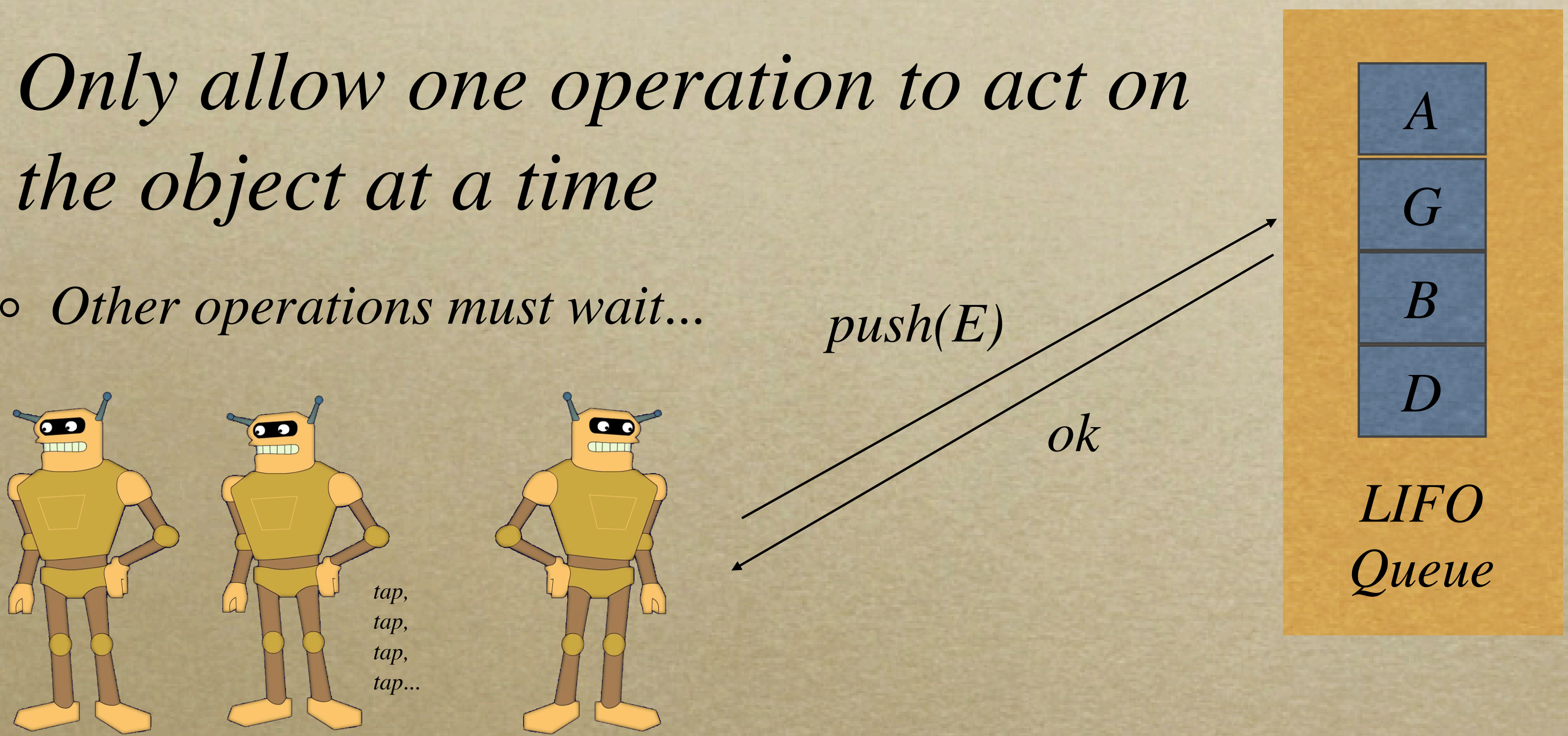
# Concurrent Data Structures

- *A* *concurrent object* *is a data structure shared by concurrent processes.*



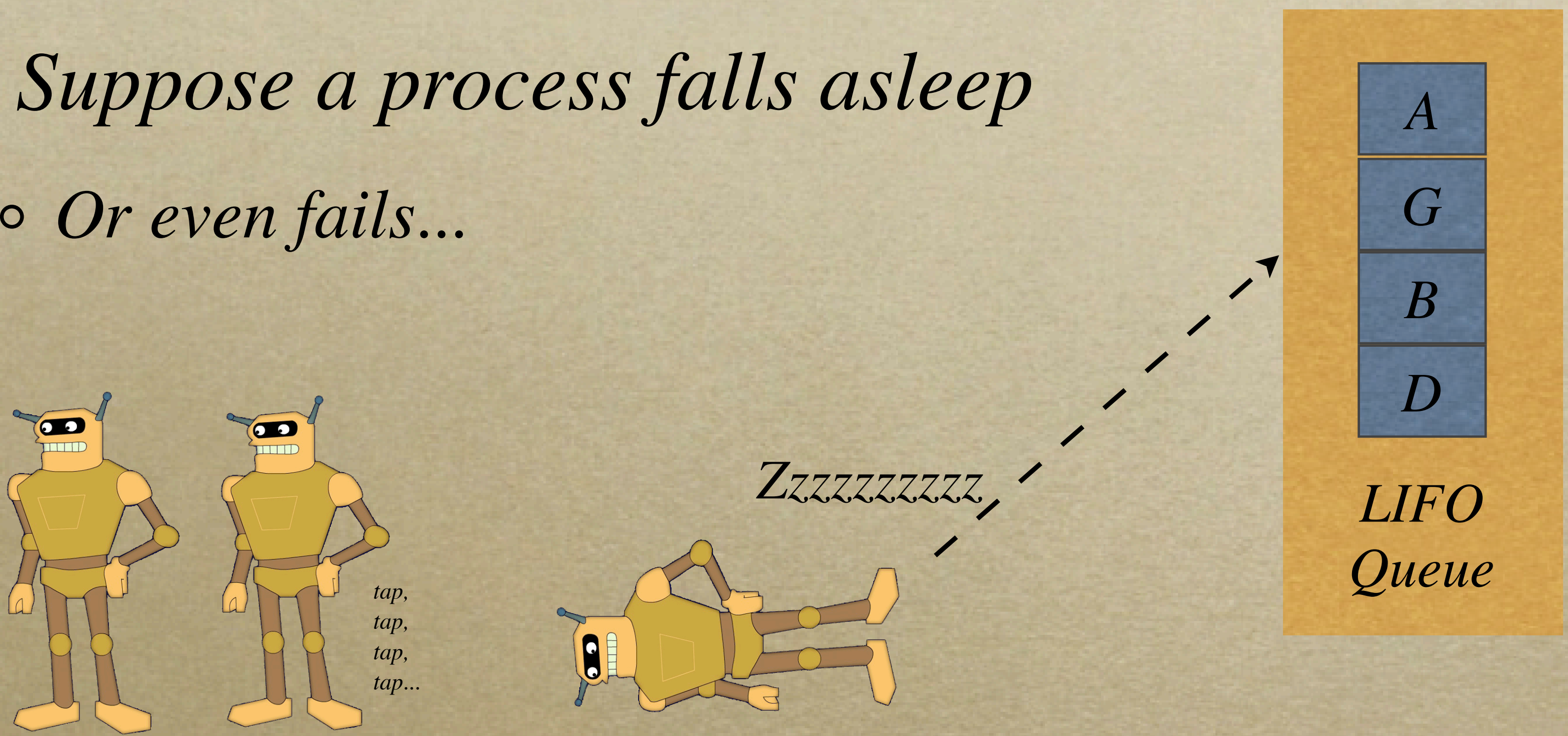
# Mutual Exclusion

- *Only allow one operation to act on the object at a time*
  - *Other operations must wait…*



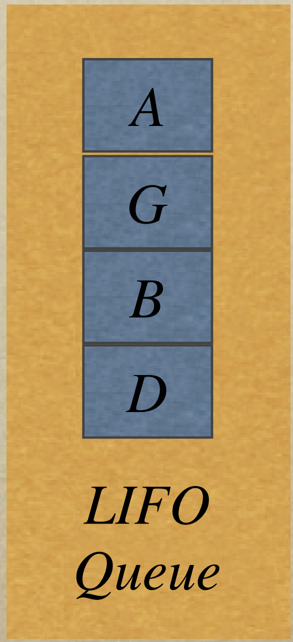
# What's Wrong With Locks?

- *Suppose a process falls asleep*
  - *Or even fails...*



# Mutual Exclusion's Problems

- *Deadlock (no progress)*
- *Priority Inversion (wrong progress)*
- *Convoying (delayed progress)*
- *Inefficiency (wasted concurrency)*
- *Undo Log (required for recovery)*



# Wait-Free Concurrent Objects

*A wait-free data structure guarantees that any process can complete any operation in a finite number of steps. (Lamport)*

# Lesser Freedoms

*A lock-free data structure guarantees that* some *process will complete an operation in a finite number of steps.*

*An obstruction-free data structure guarantees that a process will complete an operation provided there is no contention (against the operation) for a sufficient number of steps.*

# Yes, but?

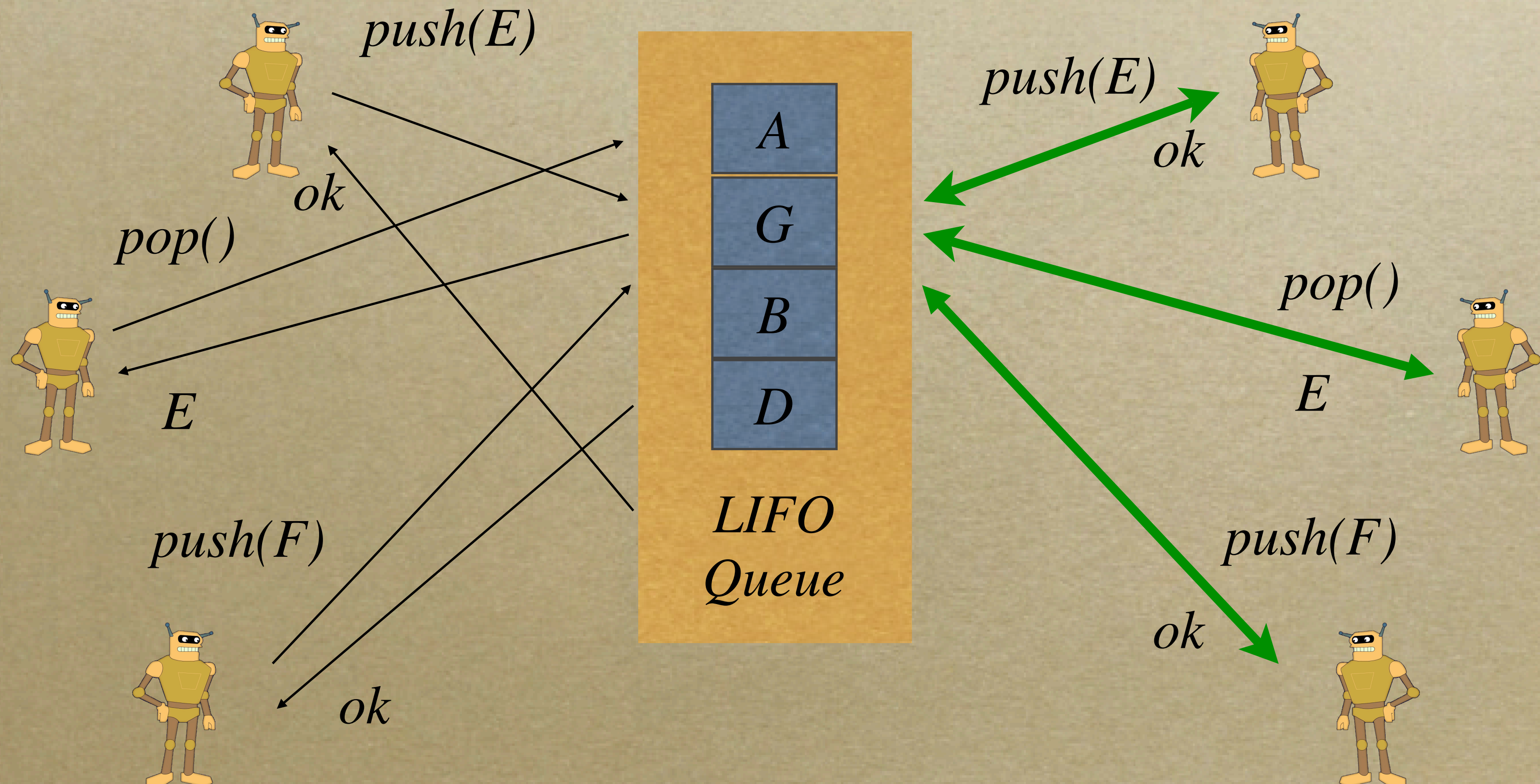
- *Can we make any object wait-free?*
  - *What primitives are necessary / sufficient for constructing wait-free objects?*
- *How do we build a wait-free object?*
  - *Is there a universal constructor?*
- *How do we know the implementation is correct?*

# You can breathe. Thank Herlihy.

- *A wait-free implementation of an object can be built out of any object with the same consensus number.*
- *There is a universal constructor using an object of infinite consensus number.*
- *We can verify correctness by showing that the implementation is linearizable.*

# Linearizability In a Nutshell

- *Each operation of the system appears to take effect instantaneously between the invocation and response.*



# Linearizability Is...

- *A local property—a concurrent system is linearizable if and only if each individual object is linearizable*
- *A non-blocking property—a total operation (that is, defined for all object states) is never required to block*

*How does this compare to sequential consistency or serializability?*

# Concurrency Model (Formal)

- *A concurrent system { $P_1$ ... $P_n$ ; $O_1$ ... $O_m$ } is a set of processes, $P_i$, and objects, $O_i$*
- *Processes and Objects are I/O automata with the following events:*
  - *INVOKE(P, op, O)*—op *is an operation of O*
  - *RESPOND(P, res, O)*—res *is a result value*
- *An object's operations must be total*
  - *If the object has a pending operation there is a matching enabled response*

# An I/O Automaton

- *An I/O automaton A consists of:*
  1. *States(A) – a finite or infinite set of states*
  2. *In(A) – a set of input events (always enabled)*
  3. *Out(A) – a set of output events*
  4. *Int(A) – a set of internal events*
  5. *Step(A) – a transition relation of triples (s', e, s)*
- *An event e is enabled in state s' if (s', e, s) in Step(A) for some s.*
- *A history is a sequence of enabled events starting at an initial state.*
- *I/O automata can be composed if they are compatible (that is, they share no output or internal events)*

# Implementations

- *An* implementation *of an object A is a concurrent system* $\{F_1 \ldots F_r ; R\}$
  - *the* $F_i$ *are called* front-ends
  - *object R is called the* representation object
- *The external events of the implementation are just the external events of A.*
- *The* $F_i$ *share no events; they only communicate through R*

# A Consensus Protocol is...

*A concurrent system { $P_1$ ... $P_n$ ; $X_1$ ... $X_m$ } of n processes, where*

- *each $P_i$ starts with an input value from some domain D*
- *the $P_i$ communicate by applying operations to the objects $X_i$*
- *the processes eventually agree on a common input value and halt*

*Required to be*

- *consistent—distinct processes never decide on distinct values*
- *valid—the common decision value is the input to some process*
- *wait-free—each process decides after a finite number of steps*

# The Consensus Number

- *The* consensus number *of a concurrent object* X *is the largest* n *for which there exists a consensus protocol* $\{ P_1 \ldots P_n ; W, X \}$
  - *W is a set of read/write registers*
  - *W and X can be initialized to any state*
- *If no largest* n *exists, the consensus number is said to be infinite*

# Consensus Number Antics

- *Theorem: If X has consensus number n, and Y has consensus number m < n, then there exists no wait-free implementation of X by Y in a system of more than m processes.*
- *The theorem implies that there is a hierarchy where each level* n *of the hierarchy contains concurrent objects with consensus number* n

# Theorem Proof

*By contradiction. Assume X has consensus number n, and Y has consensus number $m < n$. Let $k > m$, assume for contradiction that $X = \{ G_1 \dots G_k ; Y \}$ has consensus number k.*

- $\{ P_1 \dots P_k ; W, X \}$ *is a consensus protocol*
- $\{ P_1 \dots P_n ; W, \{ G_1 \dots G_n ; Y \} \}$ *is wait-free*
- $\{ P_1 \cdot G_1 \dots P_n \cdot G_n ; W, Y \}$ *is a consensus protocol because composition is associative*

# Herlihy's Wait-Free Hierarchy

| Consensus Number | Object |
|---|---|
| 1 | atomic read/write registers |
| 2 | test&set, fetch&add |
| 2n - 2 | n-register assignment |
| ∞ | compare&swap, FIFO queue w/ peek |

# Compare-and-Swap

atomically

```
val CAS( val* addr, val old, val new)
{
   val prev = *addr;
   if (prev == old) { *addr = new; }
   return prev;
}
```

- *CMPXCHG (with "lock") – Intel x86*
- *Load Linked / Store Conditional – MIPS, PowerPC*

# Compare&Swap Register

*Theorem: A CAS register has infinite consensus number.*

```
value_t decision = ⊥;
value_t decide( value_t input) {
    first = CAS( &decision, ⊥, input);
    if ( first == ⊥ )  // CAS succeeded?
        return input;
    else
        return first;
}
```

# Wait-Free Synchronization
# Lecture 2

*CS380D—Distributed Computing*
*The University of Texas at Austin*

# Summary so far...

- *Wait-free synchronization provides guaranteed progress to all correct processes*
- *There is a wait-free hierarchy determined by an object's consensus number*
- *Compare&Swap is a universal primitive and thus can be used to implement any wait-free object*

# Hierarchy Redux

- *The wait-free hierarchy states that an object with consensus number* n *(and some registers) cannot be used to implement an object of consensus number* m > n
- *How useful is this wait-free hierarchy?*
  - *Can we use* multiple *objects of consensus number* n *to implement a higher object?*

# A Robust Hierarchy

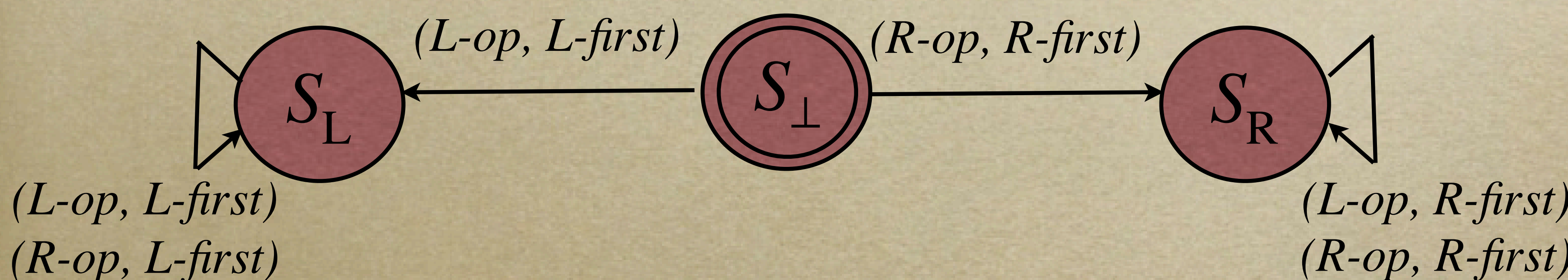
- *A robust hierarchy requires that an object at level* n *be impossible to implement using any set composed of objects at level* n-1 *or lower*
  - *Such a hierarchy would imply that there are no clever ways to combine inferior objects to create superior objects*
  - *Is Herlihy's wait-free hierarchy robust?*

# The wait-free hierarchy is not robust.

- *The* weak-sticky *object has the property that k objects (together with read/write registers) can implement an object with consensus number k+1 (Jayanti)*
- *The* weak-sticky *object is based upon the sticky bit object that solves 1-bit consensus*

# Plotkin's Sticky Bit

*State diagram:*




- *The sticky bit provides a global order for the first operation only*

# Plotkin's Sticky Bit

- *1-bit consensus object using $O_s$:*

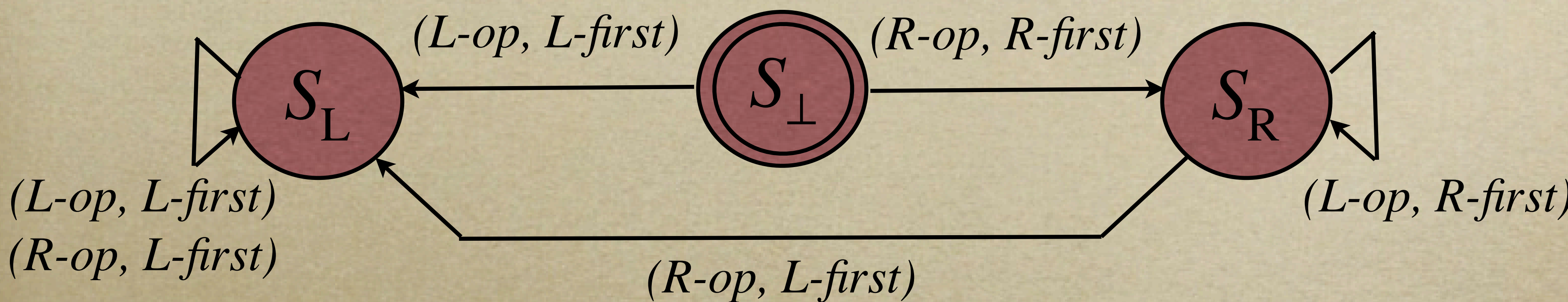
*($O_s$ is a sticky bit object,*

*$O_n$ is an n-process consensus object)*

```
Apply(P_i, propose b_i, O_n)
  return ( L-first = Apply(P_i, b_i = 1 ? L-op : R-op, O_s) )
```

*Why does the sticky bit object have consensus number ∞?*

# Weak-Sticky Objects




- *Asymmetric version of sticky bit object*
  - *Second R-op locks bit to L-first result*
  - *Provides a global order to first operation for only 2 participants*
  - *Asymmetry prevents consensus number ∞*

# Weak-Sticky Objects

- *Consensus object with $\{O_{ws}, L, R\}$:*

  *($O_{ws}$ is a weak-sticky object, L & R are shared registers,*

  *$O_2$ is a 2-process consensus object)*

| | |
|---|---|
| Apply($P_0$, propose $v_0$, $O_2$)<br>L := $v_0$<br>w := Apply($P_0$, L-op, $O_{ws}$)<br>if w = L-first<br>  return (L)<br>else<br>  return (R) | Apply($P_1$, propose $v_1$, $O_2$)<br>R := $v_1$<br>w := Apply($P_1$, R-op, $O_{ws}$)<br>if w = L-first<br>  return (L)<br>else<br>  return (R) |

*Is this valid? agreeable? wait-free?*

# Building Consensus

- *Consensus object with $\{O_{ws}, O_{n-1}, L, R\}$:*

*($O_{ws}$, L, R as before, $O_{n-1}$ is an (n-1)-process consensus object, $O_n$ is an n-process consensus object)*

```
Apply(P_i, propose v_i, O_n) {0<i<n}
L := Apply(P_i, propose v_i, O_{n-1})
w := Apply(P_i, L-op, O_ws)
if w = L-first
  return (L)
else
  return (R)
```

```
Apply(P_n, propose v_n, O_n)
R := v_n
w := Apply(P_n, R-op, O_ws)
if w = L-first
  return (L)
else
  return (R)
```

*Can other objects be used similarly to build objects with higher consensus number?*

# Universal Objects

- *An object is universal if it can be used to construct a wait-free implementation of any object (that is, it has consensus number ∞).*
- *In a system of n processes, an object is universal if and only if the object has consensus number n.*
- *CAS has consensus number ∞ and thus is a universal object.*

# How do we build using CAS?

- *Use CAS to guarantee consistency during concurrent operations*
  - *CAS can ensure that the update that succeeds is consistent with the previous view of the object*
- *Wait-freedom seems to require helping to guarantee progress to all threads*
  - Disjoint-set parallel *algorithms only help operations of other threads that "conflict"*

# A Simple Stack Object

```
struct elem {
  elem *link;
  any  data;
}

elem* qhead;
```

```
Push(elem *x)
  do
    old = qhead;
    x->link = old;
    new = x;
    cc = CAS(qhead, old, new);
  until (cc == old);
```

*Is this solution:*

*wait-free?*

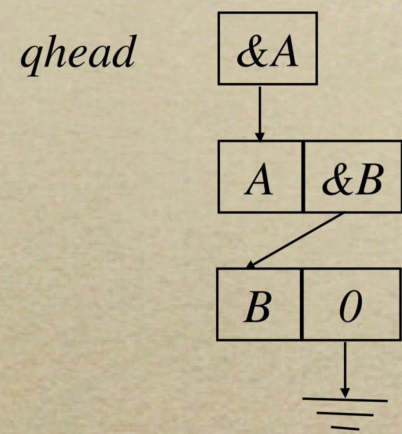
*lock-free?*

*obstruction-free?*

```
Pop()
  do
    old = qhead;
    new = old->link;
    cc = CAS(qhead, old, new);
  until (cc == old);
  return old;
```

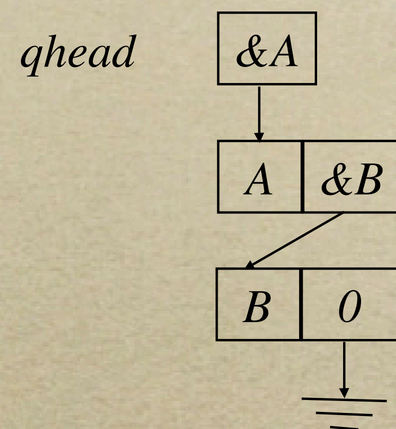
# Is this implementation correct?

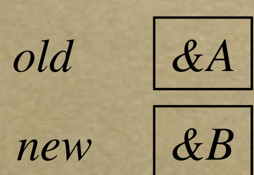
```
Pop()
do
  old = qhead;
  new = old->link;
  cc = CAS(qhead, old, new);
until (cc == old);
return old;
```



# Is this implementation correct?

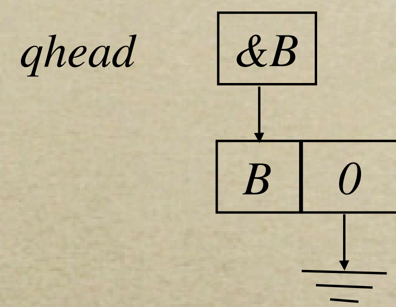
```
 Pop()
 do
   old = qhead;
   new = old->link;
→  cc = CAS(qhead, old, new);
 until (cc == old);
 return old;
```

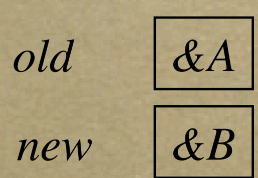


# Is this implementation correct?

```
Pop()
do
  old = qhead;
  new = old->link;
  cc = CAS(qhead, old, new);
until (cc == old);
return old;
```



# Is this implementation correct?

```
 Pop()
 do
   old = qhead;
   new = old->link;
→  cc = CAS(qhead, old, new);
 until (cc == old);
 return old;
```

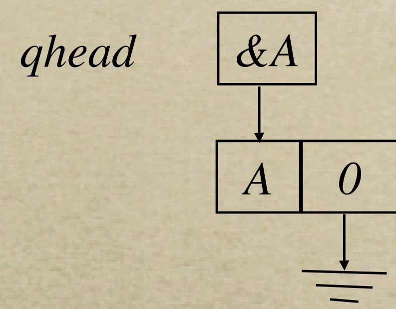
*qhead* | *0*

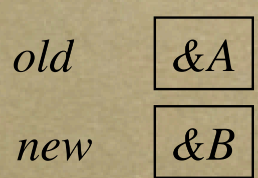
*old* | *&A*

*new* | *&B*

# Is this implementation correct?

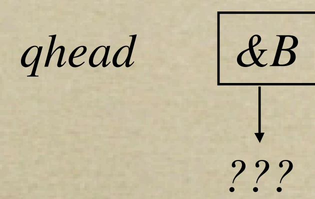
```
  Pop()
  do
    old = qhead;
    new = old->link;
→   cc = CAS(qhead, old, new);
  until (cc == old);
  return old;
```



# Is this implementation correct?

```
Pop()
do
  old = qhead;
  new = old->link;
  cc = CAS(qhead, old, new);
until (cc == old);
return old;
```




- *This is called the "ABA problem"*
- *How do we solve the problem?*

# How can we handle ABA?

*Use CAS2 (often called DCAS)*

*atomically*

```
boolean CAS2( val*  addr1,  val* addr2,
        val old1, val old2, val  new1, val new2) {
    if (*addr1 == old1 && *addr2 == old2) {
        *addr1 = new1;  *addr2 = new2;
        return true;
    } else
        return false;
}
```

*How is this useful?*

# Proposed Stack with CAS2

```
Pop()
  do
    old = qhead;
    link_field = &(old->link);
    new = *link_field;
    cc = CAS2(qhead, *link_field,
              old, new, new, new);
  until (cc);
  return old;
```

- *Does it make sense to CAS(link, new, new)?*
- *Is there a more general way to ensure the object hasn't changed?*

# DWCAS

- *Double-wide CAS (DWCAS) performs a CAS on memory locations comprised of two adjacent words*
- *Adding a version number to each location ensures the location is the expected version*
  - *Update version number when object is modified*
  - *Included expected version number in DWCAS*

# Stack with DWCAS

```
struct elem {
  elem *link;
  any data;
}
```

```
struct qhead {
  elem *link;
  int seq;
} qhead;
```

```
Push(elem *x)
  do
    old = qhead.link;
    oldseq = qhead.seq;
    x->link = old;
    cc = DWCAS(qhead,
               <old, oldseq>,
               <x, oldseq+1>);
  until (cc);
```

```
Pop()
  do
    old = qhead.link;
    oldseq = qhead.seq;
    new = old->link;
    cc = DWCAS(qhead,
               <old, oldseq>,
               <new, oldseq+1>);
  until (cc);
  return old;
```

# Problems with DWCAS

- *Requires more memory per object*
- *Does it* really *solve the problem?*
    - *Version representation is finite*
- *Assumes type-stable memory*
    - *Restricts reuse to same type*
- *We need lock-free garbage collection*
    - *Allows for simpler implementation of lock-free objects*

# Lock-free Garbage Collection

- *We have to address the garbage collection problem while maintaining lock-free access to data structures*
- *How do we tell whether an object is really garbage?*
- *How do we track memory using lock-free data structures?*

# Hazardous References

- *A* hazardous reference *is an address that without further validation can be used to access a node after it has been deleted*
- *Hazard pointers are used for each thread to track hazardous references*

old *is hazardous*

```
Pop()
do
  old = qhead;
  new = old->link;
  cc = CAS(qhead, old, new);
until (cc == old);
return old;
```

# Safe Memory Reclamation

- *Safe Memory Reclamation (SMR)*

*Whenever a thread holds a hazardous reference, it must guarantee that a hazard pointer contains the reference (Michael)*

# SMR Stack Object

- *HP is shared array of hazard pointers*
- *hp = &HP[p] (where p is current thread)*

*Validity check* → (points to the `if (old != qhead)` line)

```
Pop( )
  do
    old = qhead;
    *hp = old;
    if (old != qhead) { continue; }
    new = old->link;
    cc = CAS(qhead, old, new);
  until ( cc != old );
  *hp = NULL;
  return old;
```

# When is memory “garbage”?

- *A node can queue an object for deletion when the object is semantically dead*
    - *When the object is not reachable using the current state of all other objects*
    - *In lieu of a call to* `free`*( object )*

# Memory Deallocation

- *Periodically check the list of queued deletes to see whether any hazardous references exists*
- *If no hazard pointers contain an object in the list, the object's memory may safely be deallocated*

*How can we tell if there are hazard pointers to the object?*

# SMR Algorithm

```
// constants
int R; // batch size
int N; // # hazard ptrs

// shared variables
Node *HP[N] = { NULL, };

// static private vars
int dcount = 0;
Node *dlist[R];
```

```
SMR_free( Node *n )
  dlist[dcount++] = n;
  if( R == dcount )
    Scan();
```

```
Scan( )
  // Stage 1: copy hp
  < copy HP to local plist >
  // Stage 2: sort for search
  < sort plist >
  // Stage 3: free garbage
  for( i=0; i<R; ++i )
    if( find(dlist[i],plist) )
      *(new_dlist++) = dlist[i];
    else
      free( dlist[i] );
  // Stage 4: save remainder
  < copy new_dlist to dlist >
```

# Multiple Hazard Pointers

- *The HP array is scanned non-atomically, requiring hazard pointers to be maintained in the same order that the HP array is scanned*

```
Dequeue( )
  while( true )
    h = qhead;
    *hp0 = h;
    if( h != qhead ) { continue; }
    t = qtail;
    next = h->link;
    *hp1 = next;
    if( h != qhead )
      { *hp0 = NULL; return EMPTY; }
    if( h == t )
      { CAS(&qtail, t, next); continue;
    data = next->data;
    if( h == CAS(&qhead, h, next) )
      break;
  *hp0 = NULL; *hp1 = NULL;
  SMR_free( h );
  return data;
```

# SMR Questions

- *Does SMR really solve ABA?*
- *Is SMR really wait-free?*

# SMR Answers

- *Does SMR really solve ABA?*
  - *SMR addresses only reallocation*
- *Is SMR really wait-free?*
  - *The operations complete in finite steps*
  - *Memory is not guaranteed to be deallocated*