

DD245 I
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

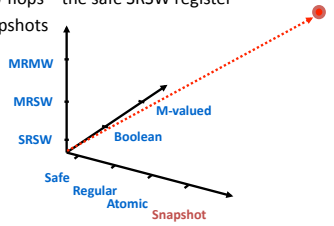
Lecture 4
Consensus, I

Mads Dam
Autumn/Winter 2011

Slides: Much material due to M. Herlihy and R Wattenhofer

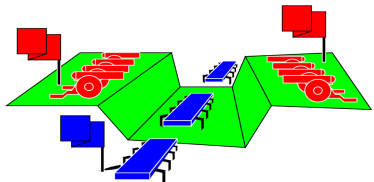
Last Lecture

- Shared memory computability:
 - What can be computed on a shared memory multiprocessor **without using locks**?
- Start with simple flip-flops – the safe SRSW register
- End with atomic snapshots
- Are we done?



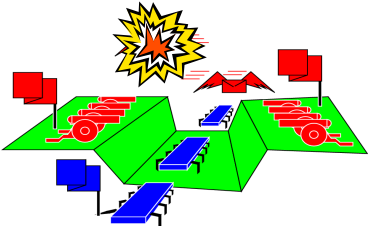
The Two Generals

- Red army wins if both sides attack simultaneously
- Red army can communicate by sending messages...



Problem: Unreliable Communication

- ... such as "lets attack tomorrow at 6am" ...
- ... but messages do not always make it!
- Task: Design a "red army protocol" that works despite message failures!



Theorem

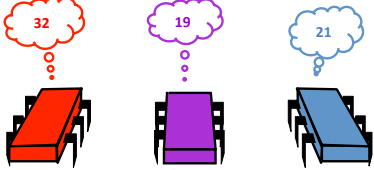
Theorem: There is no non-trivial protocol that ensures that the red armies attack simultaneously

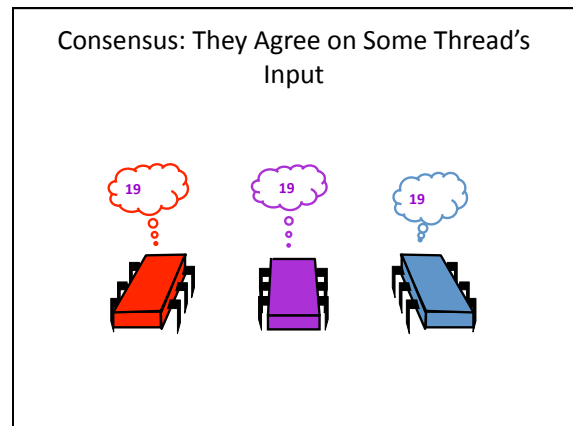
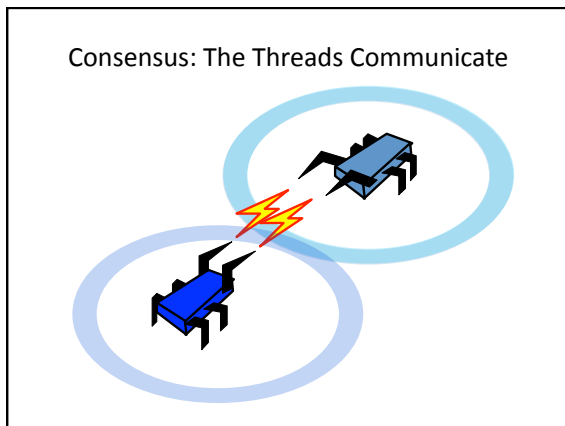
Proof:

- Consider the protocol that sends the fewest messages
- It still works if the last message is lost
- So just don't send it (messengers' union happy!)
- But now we have a shorter protocol!
- Contradicting #1

Fundamental limitation: We need an unbounded number of messages, otherwise it is possible that no attack takes place!

Consensus: Each Thread has a Private Input





Consensus is Important

With consensus, you can implement anything you can imagine...

- Decide on a leader,
- Implement mutual exclusion
- Solve the two generals problem
- Much more...

We will see that in some models, consensus is possible, in some other models, it is not

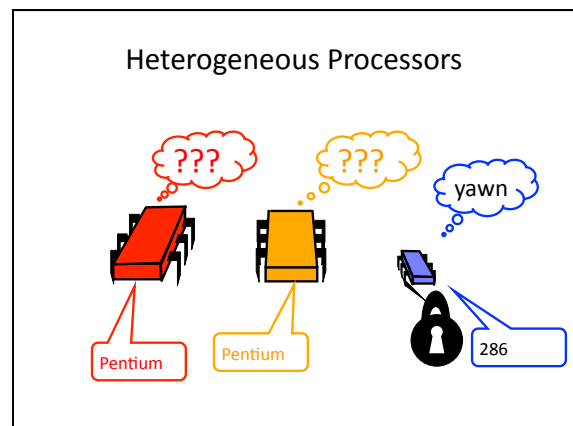
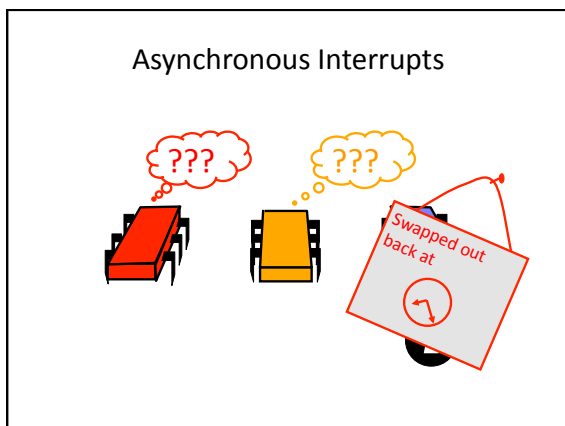
The goal is to learn whether for a given model consensus is possible or not ... and prove it!

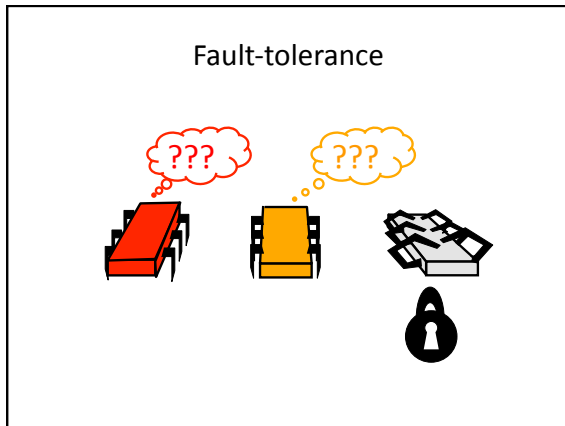
Consensus #1: Shared Memory

Protocol:

- There is a designated memory cell c .
- Initially c is in a special state “?”
- Processor 1 writes its value v_1 into c , then decides on v_1 .
- A processor $j \neq 1$ reads c until j reads something else than “?”, and then decides on that.

- Problems with this approach?





Consensus #2: Wait-free Shared Memory

- $n > 1$ processors
- Processors can atomically *read* or *write* (not both) a shared memory cell
- Processors might crash (stop... or become very slow...)

Wait-free implementation:

- Every process (method call) completes in a finite number of steps
- Implies that locks cannot be used → The thread holding the lock may crash and no other thread can make progress
- We assume that we have wait-free atomic registers (that is, reads and writes to same register do not overlap)

Impossibility of Consensus

Theorem There is no wait-free implementation of n -thread consensus, $n > 1$, from atomic read-write registers

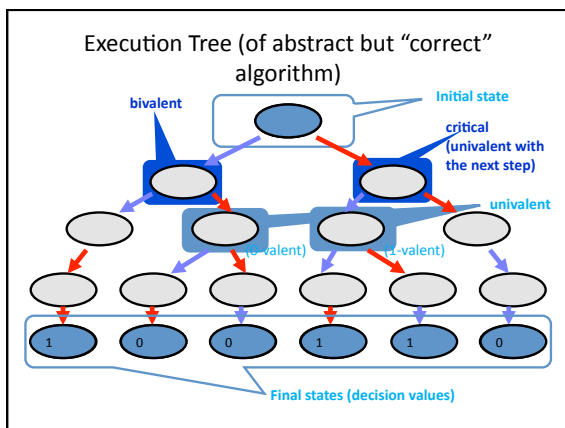
Credits: Fischer-Lynch-Paterson, JACM 1985

Proof strategy:

- Assume a protocol exists
- Reason about its properties
- Derive a contradiction
- QED

Proof

- Make it simple
 - There are only two threads **A** and **B** and the input is binary
- Assume that there is a protocol
- In this protocol, either **A** or **B** “moves” in each step
- Moving means
 - Register read
 - Register write



Bivalent vs. Univalent

- Wait-free computation is a tree
- Bivalent system states
 - Outcome is not fixed
- Univalent states
 - Outcome is fixed
 - Maybe not “known” yet
 - 1-valent and 0-valent states
- **Claim**
 - Some initial system state is bivalent
 - This means that the outcome is not always fixed from the start

Some Initial State Is Bivalent

Claim: Every 2 thread protocol has a bivalent initial state

Proof:
 Consider this initial state

All executions must decide same value, including these two:

QED

A Critical State Can Be Reached

Lemma: Every wait-free consensus protocol has a critical state

Proof: Some initial state is bivalent. Trace a path to critical state

Model Dependency

- So far, everything was memory-independent!
- True for
 - Registers
 - Message-passing
 - Carrier pigeons
 - Any kind of asynchronous computation
- Threads
 - Perform reads and/or writes
 - To the same or different registers
 - Possible interactions?

Possible Interactions

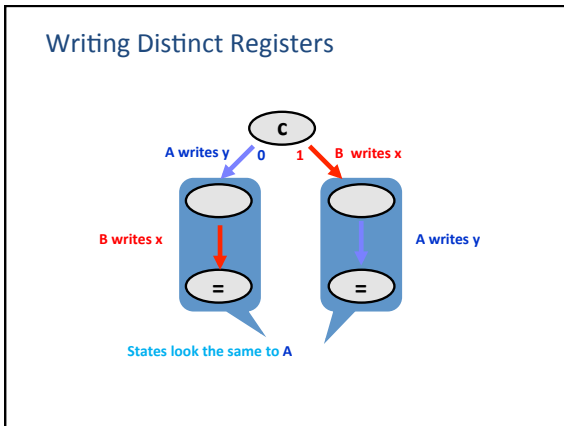
	A reads x			
	x.read()	y.read()	x.write()	y.write()
x.read()	?	?	?	?
y.read()	?	?	?	?
x.write()	?	?	?	?
y.write()	?	?	?	?

B writes y

Reading Registers

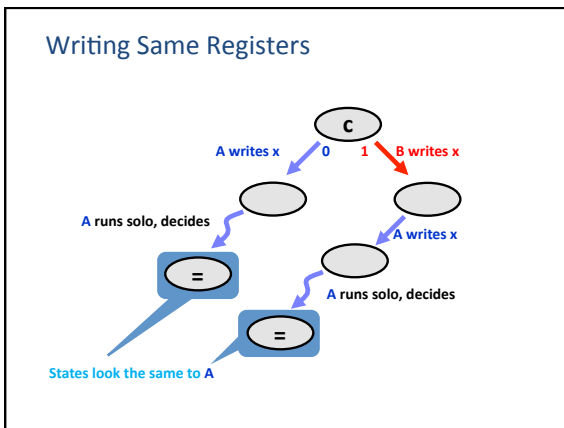
Possible Interactions

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	?
y.write()	no	no	?	?



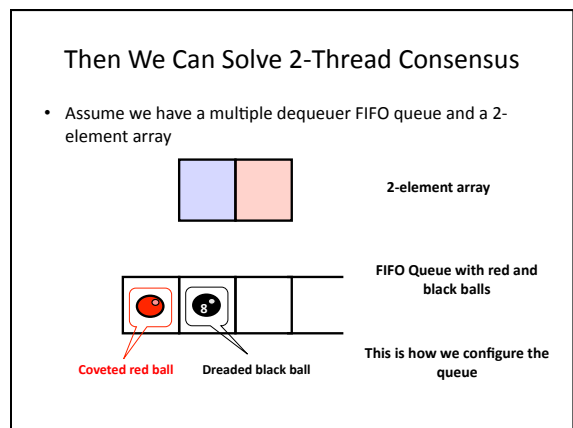
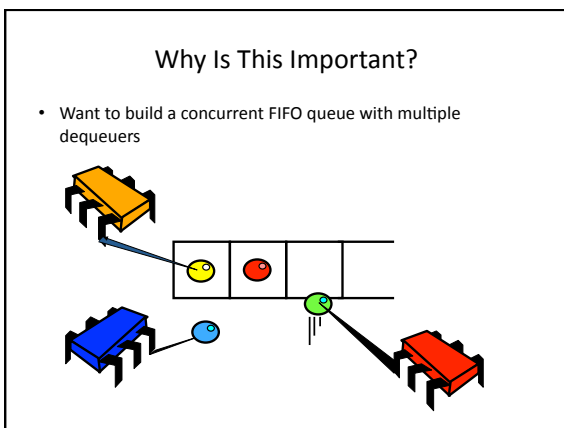
Possible Interactions

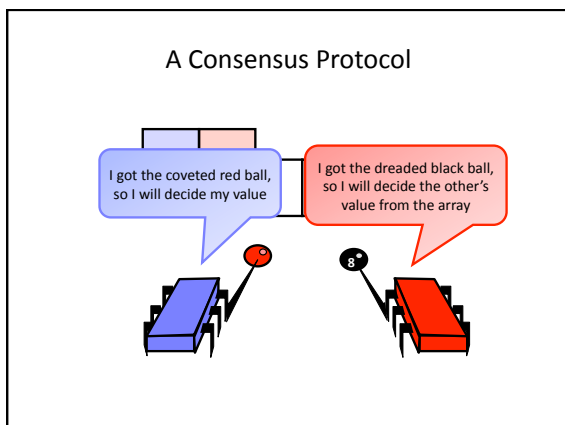
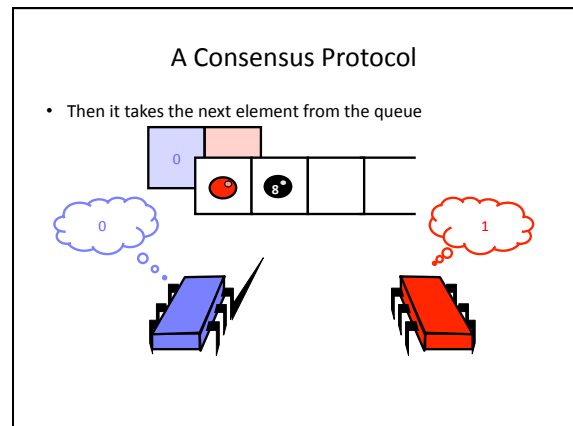
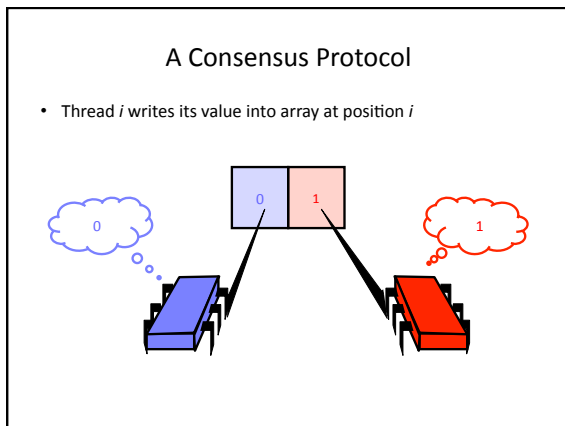
	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	?	no
y.write()	no	no	no	?



That's All, Folks!

	x.read()	y.read()	x.write()	y.write()
x.read()	no	no	no	no
y.read()	no	no	no	no
x.write()	no	no	no	no
y.write()	no	no	no	no





A Consensus Protocol

Why does this work?

- If one thread gets the red ball, then the other gets the black ball
- Winner can take its own value
- Loser can find winner's value in array
 - Because threads write array before dequeuing from queue

Implication

- We can solve 2-thread consensus using only
 - A two-dequeuer queue
 - Atomic registers

Implication

- Assume there exists
 - A queue implementation from atomic registers
- Given
 - A consensus protocol from queue and registers
- Substitution yields
 - A wait-free consensus protocol from atomic registers

Contradiction

Corollary

- It is *impossible* to implement a two-dequeuer wait-free FIFO queue with read/write shared memory.
- This was a proof by reduction; important beyond NP-completeness...

Consensus #3: Read-Modify-Write Shared Memory

- $n > 1$ processors
- Wait-free implementation
- Processors can atomically read *and* write a shared memory cell in one atomic step
- The value written can depend on the value read
- We call this a read-modify-write (RMW) register
- Can we solve consensus using a RMW register...?

Consensus Protocol Using an RMW Register

- There is a cell c , initially $c = "?"$
- Every processor i does the following

```

if (c == "?") then
    write(c, vi);
else
    decide c;
    
```

atomic step

RMW(c)

Discussion

- Protocol works correctly
 - One processor accesses c first; this processor will determine decision
- Protocol is wait-free
- RMW is quite a strong primitive
 - Can we achieve the same with a weaker primitive?

Read-Modify-Write

```

public class RMW {
    private int value;

    public int synchronized rmw(function f) {
        int prior = this.value;
        this.value = f(this.value);
        return prior; } }
    
```

- Ingredients:
 - Location c
 - Function f
- Operation:
 - Replaces value x of c with $f(x)$
 - Returns value x of cell location c

Read-Modify-Write: Read

```

public class RMW {
    private int value;

    public synchronized int read() {
        int prior = this.value;
        this.value = this.value;
        return prior;
    }
}
    
```

Identify function

Read-Modify-Write: Test&Set

```

public class RMW {
    private int value;

    public synchronized int TASO() {
        int prior = this.value;
        this.value = 1;
        return prior;
    }
}
    
```

Constant function

Read-Modify-Write: Fetch&Inc

```

public class RMW {
    private int value;

    public synchronized int FAIO() {
        int prior = this.value;
        this.value = this.value+1;
        return prior;
    }
}
    
```

Increment function

Read-Modify-Write: Fetch&Add

```
public class RMW {
    private int value;

    public synchronized int FAA(int x) {
        int prior = this.value;
        this.value = this.value+x;
        return prior;
    }
}
```

Addition function

Read-Modify-Write: Swap

```
public class RMW {
    private int value;

    public synchronized int swap(int x) {
        int prior = this.value;
        this.value = x;
        return prior;
    }
}
```

Old value

Read-Modify-Write: Compare&Swap

```
public class RMW {
    private int value;

    public synchronized int CAS(int old, int new) {
        int prior = this.value;
        if(this.value == old)
            this.value = new;
        return prior;
    }
}
```

"Complex" function

Definition of Consensus Number

- An object has **consensus number n**
 - If it can be used
 - Together with atomic read/write registers
 - To implement n -thread consensus, but not $(n+1)$ -thread consensus
- Example:
 - Atomic read/write registers have consensus number 1
 - Works with 1 process
 - We have shown impossibility with 2

Consensus Number Theorem

Theorem If you can implement X from Y and X has consensus number c , then Y has consensus number at least c

- Consensus numbers are a useful way of measuring synchronization power
- An alternative formulation:
 - If X has consensus number c
 - And Y has consensus number $d < c$
 - Then there is no way to construct a wait-free implementation of X by Y
- This theorem will be very useful
 - Unforeseen practical implications!

Theorem

- A RMW is *non-trivial* if there exists a value v such that $v \neq f(v)$
 - Test&Set, Fetch&Inc, Fetch&Add, Swap, Compare&Swap, general RMW...
 - But not read

Theorem Any non-trivial RMW register has consensus number at least 2

- Implies no wait-free implementation of RMW registers from read/write registers
- Hardware RMW instructions not just a convenience

Proof

- A two-thread consensus protocol using any non-trivial RMW object:

```

public class RMWConsensusFor2 implements Consensus{
    private RMW r; // Initialized to v

    public Object decide() {
        int i = Thread.myIndex();
        if(r.rmw(f) == v) // Am I first?
            return this.announce[i]; // Yes, return my input
        else
            return this.announce[1-i]; // No, return other's input
    }
}
    
```

Interfering RMW

Let F be a set of functions such that for all f and g in F :

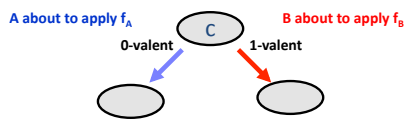
- Either they commute: $f(g(x))=g(f(x))$
- Or they overwrite: $f(g(x))=f(x)$

Claim Any such set of RMW objects has consensus number exactly 2

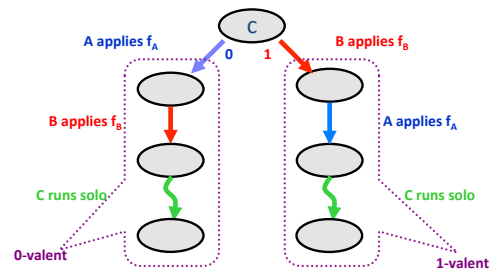
- Examples:**
- Overwrite
 - Test&Set, Swap
 - Commute
 - Fetch&Inc

Proof

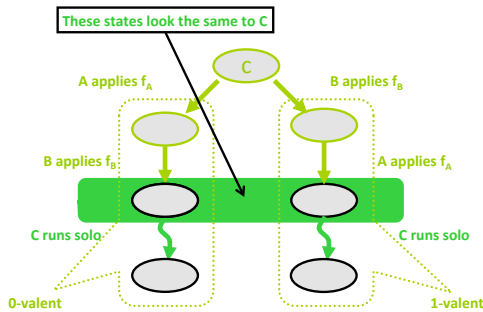
- There are three threads, A, B, and C
- Consider a critical state c:



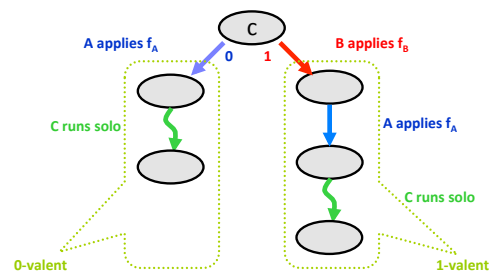
Proof: Maybe the Functions Commute

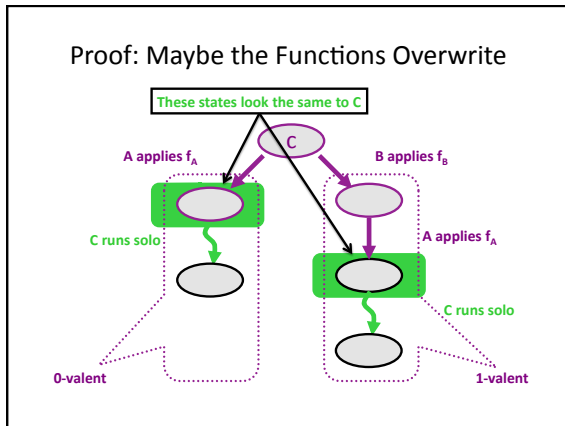


Proof: Maybe the Functions Commute



Proof: Maybe the Functions Overwrite





Impact

- Many early machines used these “weak” RMW instructions
 - Test&Set (IBM 360)
 - Fetch&Add (NYU Ultracomputer)
 - Swap (original SPARC)
- We now understand their limitations

Consensus with Compare&Swap

```

public class RMWConsensus implements Consensus {
    private RMW r; // Initialized to -1

    public Object decide() {
        int i = Thread.myIndex();
        int j = r.CAS(-1, i); // Am I first?
        if (j == -1) // Yes, return my input
            return this.announce[i];
        else // No, return other's input
            return this.announce[j];
    }
}
    
```

