Lecture 7: progress guarantees, concurrent operations hierarchy, consensus number

obstruction-free, lock-free, wait-free, safe register, regular register, atomic register, register snapshot, consensus number

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l7.pdf

In previous episodes

"Alphabet" of concurrent execution

- Timeline, Event, Interval, Precedence

Simple "words" related to mutual exclusion:

- Mutual exclusion, Deadlock-freedom, Starvation-freedom

"Sentences" that describe proper mutual exclusion primitive:

- Peterson's algorithm
- FilterLock
- Lower bounds on the number of locations

"Grammar rules" for particular concurrent objects:

- Sequential object, sequential specification
- Concurrent object, consistency
- Linearizability, linearization points
- Non-linearizable executions

In previous episodes

"Alphabet" of concurrent execution

- Timeline, Event, Interval, Precedence

Simple "words" related to mutual exclusion:

- Mutual exclusion, Deadlock-freedom, Starvation-freedom

"Sentences" that describe proper mutual exclusion primitive:

- Peterson's algorithm
- FilterLock
- Lower bounds on the number of locations

"Grammar rules" for particular concurrent objects:

- Sequential object, sequential specification
- Concurrent object, consistency
- Linearizability, linearization points
- Non-linearizable executions

We were focusing on **correctness**. Today we will focus on **progress**.

Supplementary materials

- Chapters 1-5 in "The Art of Multiprocessor Programming"
- Companion materials `https://booksite.elsevier.com/9780123973375`

Our course briefly introduces some important theoretical concepts, for better understanding consider looking through

- `chapter_03.ppt`
- `chapter_04.ppt`
- `chapter_05.ppt`

Lecture plan

1. Progress conditions
2. Space of registers
3. Register constructions
4. Atomic snapshots
5. Consensus number
6. Summary

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds

Question time

Question: **Starvation-free** is a property of mutual exclusion algorithm. If some thread owns mutex, other thread will have to wait. Name concurrent problems that are related to this situation.

Question time

Question: **Starvation-free** is a property of mutual exclusion algorithm. If some critical section (code inside mutex) takes a lot of time then progress of system-as-a-whole will be very slow. Imagine that all critical sections are small (e.g. 100 machine instructions). Could you guarantee "fast" progress of different threads?

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns
- **Wait-free**: every thread calling a method eventually returns

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns
- **Wait-free**: every thread calling a method eventually returns

Remember: this *eventually* may heavily depend on scheduler, processor, algorithm

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns
- **Wait-free**: every thread calling a method eventually returns

	Non-blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns
- **Wait-free**: every thread calling a method eventually returns

	Non-blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Non-blocking progress conditions:

- **Wait-free**, **Lock-free**: computation as a whole makes progress independently of how the system schedules threads

Progress conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds
- **Lock-free**: some thread calling a method eventually returns
- **Wait-free**: every thread calling a method eventually returns

	Non-blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free

Non-blocking progress conditions:

- **Wait-free**, **Lock-free**: computation as a whole makes progress independently of how the system schedules threads

Dependent blocking progress conditions:

- **Starvation-free**, **Deadlock-free**: progress occurs only if the underlying platform provides certain guarantees

Progress conditions

Non-blocking progress conditions:

- **Wait-free**, **Lock-free**: computation as a whole makes progress independently of how the system scheduler threads

Dependent blocking progress conditions:

- **Starvation-free**, **Deadlock-free**: progress occurs only if the underlying platform provides certain guarantees

Progress conditions

Non-blocking progress conditions:

- **Wait-free**, **Lock-free**: computation as a whole makes progress independently of how the system scheduler threads

Dependent blocking progress conditions:

- **Starvation-free**, **Deadlock-free**: progress occurs only if the underlying platform provides certain guarantees

Dependent non-blocking progress condition:

- **Obstruction-free** method: if, from any point after which it executes in isolation, it finishes in a finite number of steps

Question time

Question: Describe **obstruction-free** but not **lock-free** method.

Reminder:

- **Lock-free** – someone makes progress even with "bad" scheduler
- **Obstruction-free** – someone makes progress if others are stopped

Progress conditions: summary

Non-blocking progress conditions: progress happens unconditionally

- lock-free: for some thread
- wait-free: for any thread

Dependent blocking progress conditions: progress happens if scheduling is "fair"

- deadlock-free: for some thread
- starvation-free: for any thread

Dependent non-blocking progress condition: progress happens if scheduling is "unfair"

- obstruction-free: for thread in isolation (other threads are frozen)

Progress conditions: summary

Non-blocking progress conditions: progress happens unconditionally

- lock-free: for some thread
- wait-free: for any thread

Dependent blocking progress conditions: progress happens if scheduling is "fair"

- deadlock-free: for some thread
- starvation-free: for any thread

Dependent non-blocking progress condition: progress happens if scheduling is "unfair"

- obstruction-free: for thread in isolation (other threads are frozen)

Theorem

Wait-free *implies* ***lock-free*** *implies* ***obstruction-free***
Starvation-free *implies* ***deadlock-free***

Lecture plan

1. Progress conditions
2. Space of registers
3. Register constructions
4. Atomic snapshots
5. Consensus number
6. Summary

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?
- What is the *weakest* shared object that allows shared-memory computation?

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?
- What is the *weakest* shared object that allows shared-memory computation?

Alan Turing showed what **is computable** and what is **not computable** on a sequential machine.

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?
- What is the *weakest* shared object that allows shared-memory computation?

Alan Turing showed what **is computable** and what is **not computable** on a sequential machine.

Turing computability

- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?
- What is the *weakest* shared object that allows shared-memory computation?

Alan Turing showed what **is computable** and what is **not computable** on a sequential machine.

Turing computability

- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Shared-Memory Computability?

Concurrency foundations

- What is the *weakest* form of communication that supports mutual exclusion?
- What is the *weakest* shared object that allows shared-memory computation?

Alan Turing showed what **is computable** and what is **not computable** on a sequential machine.

Turing computability

- Mathematical model of computation
- What is (and is not) computable
- Efficiency (mostly) irrelevant

Shared-Memory Computability?

- Mathematical model of concurrent computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

Foundations

To understand modern multiprocessors we need to ask some basic questions:

Foundations

To understand modern multiprocessors we need to ask some basic questions:

- What is the weakest useful form of shared memory?

Foundations

To understand modern multiprocessors we need to ask some basic questions:

- What is the weakest useful form of shared memory?
- What can it do?

Register

Memory location[1]

```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

[1] *register* is a historical name

Register: capacity

Memory location

```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

T is a type of register

- boolean
- m-bit integer

Register: supported level of concurrency

Concurrency: at least two threads

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)
- Multi-Reader/Single-Writer Register

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)
- Multi-Reader/Single-Writer Register (MRSW)

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)
- Multi-Reader/Single-Writer Register (MRSW)
- Multi-Reader/Multi-Writer Register

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)
- Multi-Reader/Single-Writer Register (MRSW)
- Multi-Reader/Multi-Writer Register (MRMW)

Register: supported level of concurrency

Concurrency: at least two threads

- Single-Reader/Single-Writer Register (SRSW)
- Multi-Reader/Single-Writer Register (MRSW)
- Multi-Reader/Multi-Writer Register (MRMW)
- SRMW is exotic, we will not discuss it

Register consistency: safe register

Safe Register:

- If reads and writes do not overlap returns latest value (looks like sequential register)

Register consistency: safe register

Safe Register:

- If reads and writes do not overlap returns latest value (looks like sequential register)
- Returns **arbitrary** value in case of overlap (looks like Undefined Behaviour)

Register consistency: safe register

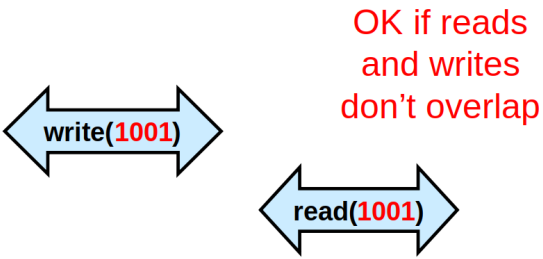
Safe Register:

- Returns *latest* value if reads and writes do not overlap
- Returns *arbitrary* (but valid) value in case of overlap but does not stop execution

Register consistency: safe register

Safe Register:

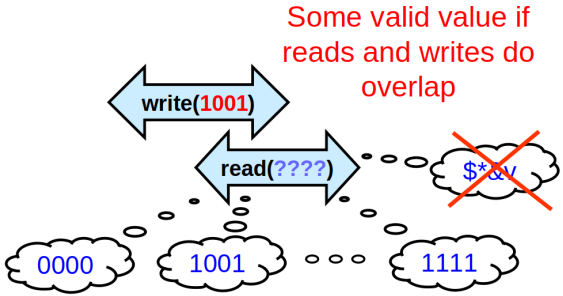
- Returns *latest* value if reads and writes do not overlap
- Returns *arbitrary* (but valid) value in case of overlap but does not stop execution



Register consistency: safe register

Safe Register:

- Returns *latest* value if reads and writes do not overlap
- Returns *arbitrary* (but valid) value in case of overlap but does not stop execution



Register consistency: regular register

Regular Register:

- Returns old value if no overlap (safe)

Register consistency: regular register

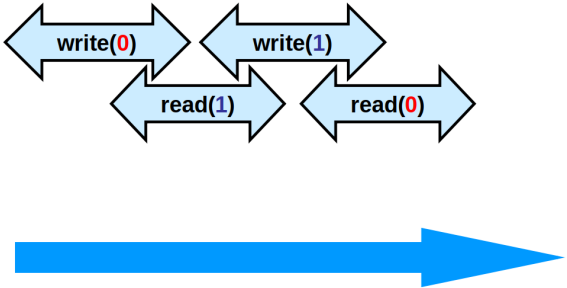
Regular Register:

- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)

Register consistency: regular register

Regular Register:

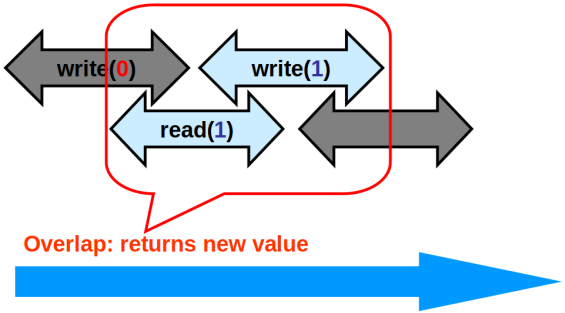
- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



Register consistency: regular register

Regular Register:

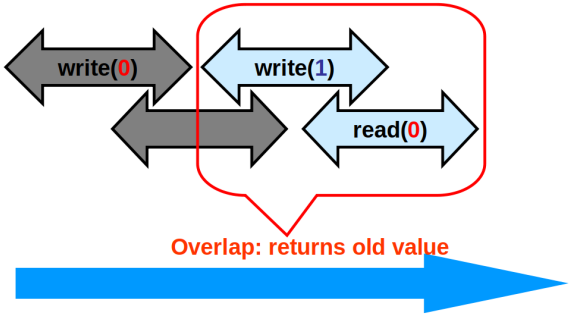
- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



Register consistency: regular register

Regular Register:

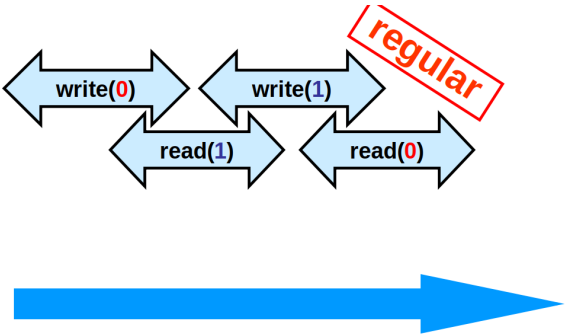
- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



Register consistency: regular register

Regular Register:

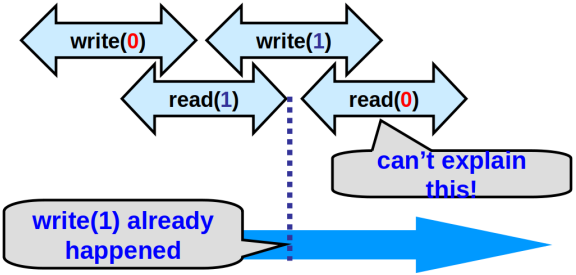
- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



Register consistency: regular register != linearizable

Regular Register:

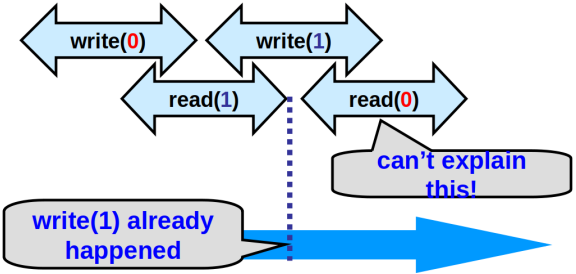
- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



Register consistency: regular register != linearizable

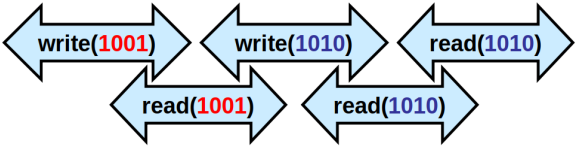
Regular Register:

- Returns old value if no overlap (safe)
- Old or one of new values if overlap (not arbitrary)



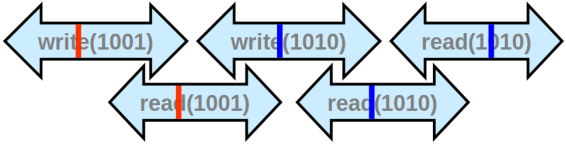
Regular != linearizable

Register consistency: atomic register



Linearizable to sequential safe register

Register consistency: atomic register



Linearizable to sequential safe register

Register consistency: jokes to help

- Register means shared memory location

Register consistency: jokes to help

- Register means shared memory location
- **Safe** register is quite unsafe – generates random values under contention

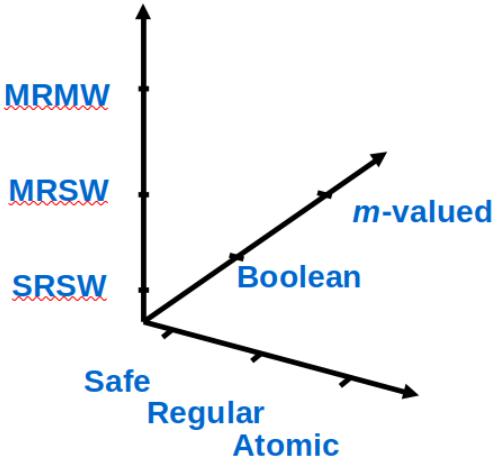
Register consistency: jokes to help

- Register means shared memory location
- **Safe** register is quite unsafe – generates random values under contention
- **Regular** register exhibits irregular behaviour – could oscillate under contention

Register consistency: jokes to help

- Register means shared memory location
- **Safe** register is quite unsafe – generates random values under contention
- **Regular** register exhibits irregular behaviour – could oscillate under contention
- **Atomic** register changes state somewhere inside method – linearization point depends on execution

Register space

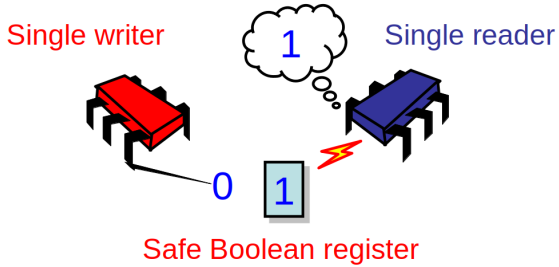


Weakest register

SRSW Safe Boolean register

Weakest register

SRSW Safe Boolean register

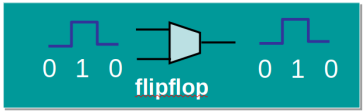


Weakest register

SRSW Safe Boolean register

Single writer

Single reader



Get correct reading if not during state transition

Spoilers

From SRSW safe Boolean register:

Spoilers

From SRSW safe Boolean register:

- All the other registers

Spoilers

From SRSW safe Boolean register:

- All the other registers
- Mutual exclusion

Spoilers

From SRSW safe Boolean register:

- All the other registers
- Mutual exclusion
- Consistent and non-blocking snapshot of N registers

Spoilers

From SRSW safe Boolean register:

- All the other registers
- Mutual exclusion
- Consistent and non-blocking snapshot of N registers

But not everything!

- Consensus hierarchy

Lecture plan

1. Progress conditions
2. Space of registers
3. Register constructions
4. Atomic snapshots
5. Consensus number
6. Summary

Register constructions: plan

Not interesting to rely on mutual exclusion in register constructions

- We want registers to implement mutual exclusion!
- It's cheating to use mutual exclusion to implement itself!

Register constructions: plan

Not interesting to rely on mutual exclusion in register constructions

- We want registers to implement mutual exclusion!
- It's cheating to use mutual exclusion to implement itself!

Definition

An object implementation is **wait-free** if every method call completes in a finite number of steps

Register constructions: plan

Not interesting to rely on mutual exclusion in register constructions

- We want registers to implement mutual exclusion!
- It's cheating to use mutual exclusion to implement itself!

Definition

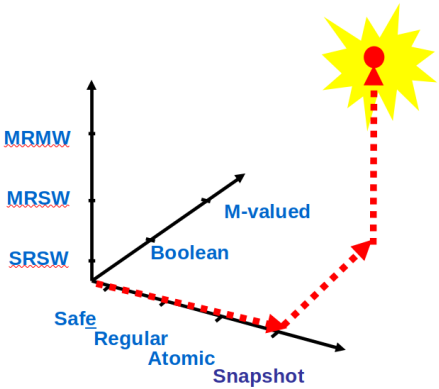
An object implementation is **wait-free** if every method call completes in a finite number of steps

No mutual exclusion

- Thread could halt in critical section
- Build mutual exclusion from registers

Register constructions: plan

From Safe SRSW Boolean to Atomic Snapshots



Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular
- MRSW Atomic
- MRMW Atomic
- Atomic snapshot

Register constructions: road map

- SRSW Safe Boolean **(1)**
- MRSW Safe Boolean **(2)**
- MRSW Regular Boolean
- MRSW Regular
- MRSW Atomic
- MRMW Atomic
- Atomic snapshot

Safe MRSW Boolean from Safe SRSW Boolean: notation

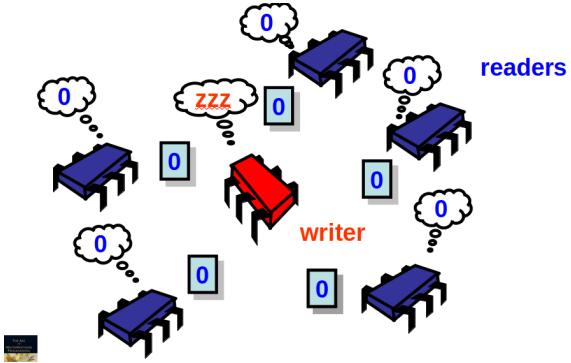
```
public class SafeBoolMRSWRegister implements Register<Boolean> {
  public boolean read() { ... }
  public void write(boolean x) { .. }
}
```

Safe MRSW Boolean from Safe SRSW Boolean: notation

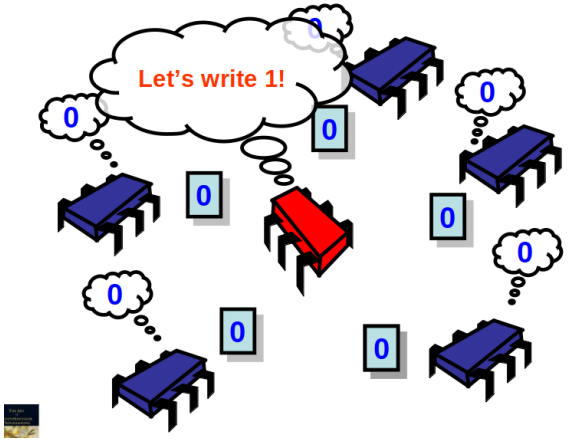
```
public class SafeBoolMRSWRegister implements Register<Boolean> {
  public boolean read() { ... }
  public void write(boolean x) { .. }
}
```

- Safe – consistency
- Bool – capacity
- MRSW – concurrency

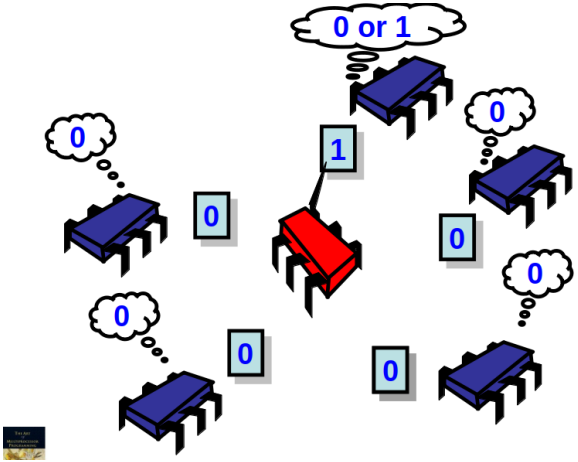
Safe MRSW Boolean from Safe SRSW Boolean: idea



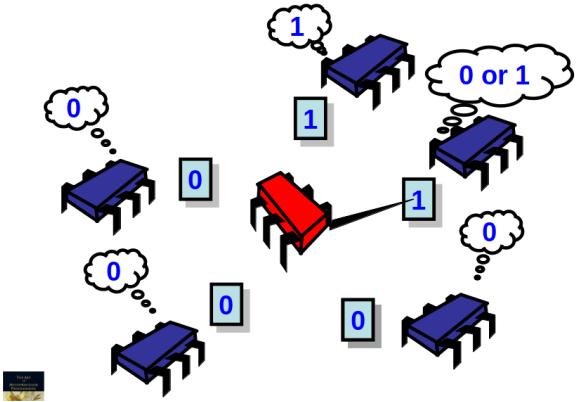
Safe MRSW Boolean from Safe SRSW Boolean: idea



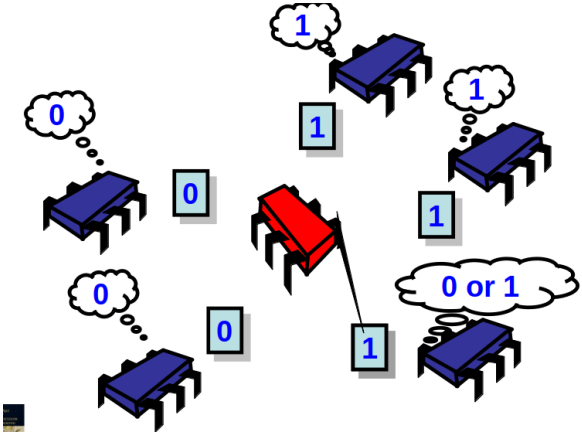
Safe MRSW Boolean from Safe SRSW Boolean: idea



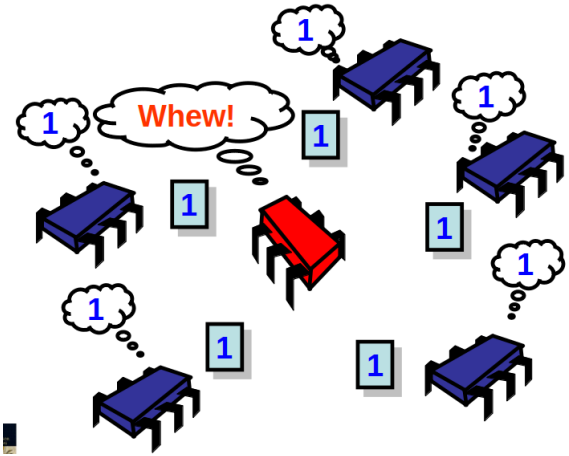
Safe MRSW Boolean from Safe SRSW Boolean: idea



Safe MRSW Boolean from Safe SRSW Boolean: idea



Safe MRSW Boolean from Safe SRSW Boolean: idea



Safe MRSW Boolean from Safe SRSW Boolean

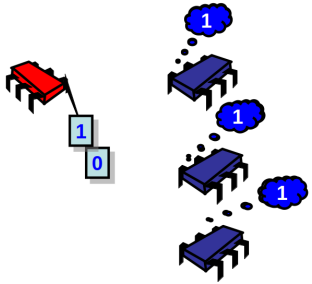
```
public class SafeBoolMRSWRegister implements Register<Boolean> {
  // use weaker registers
  private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];

  public void write(boolean x) {
    for (int j = 0; j < N; j++) {
      r[j].write(x); // announce new value for every other thread
    }
  }
  public boolean read() {
    int i = ThreadID.get();
    return r[i].read(); // read own copy
  }
}
```

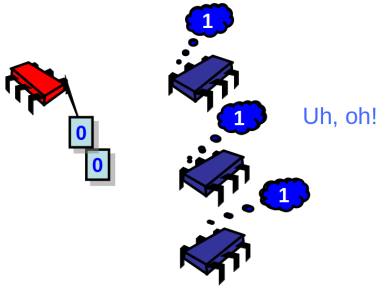
Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean **(2)**
- MRSW Regular Boolean **(3)**
- MRSW Regular
- MRSW Atomic
- MRMW Atomic
- Atomic snapshot

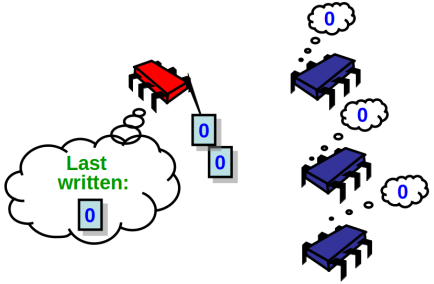
Regular MRSW Boolean from Safe MRSW Boolean: idea



Regular MRSW Boolean from Safe MRSW Boolean: idea



Regular MRSW Boolean from Safe MRSW Boolean: idea



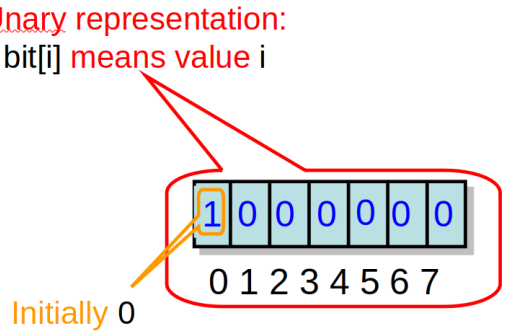
Regular MRSW Boolean from Safe MRSW Boolean: idea

```
public class RegBoolMRSWRegister implements Register<Boolean> {
  private SafeBoolMRSWRegister value; // actual data
  private thread_local boolean old; // last bit this thread wrote
  public void write(boolean x) {
    if (old != x) {   // if new value is different from my previous write
      value.write(x); // then update
      old = x;        // remember my decision
    }
  }
  public boolean read() {
    return value.read(); // in case of overlap any result (true/false) is OK
  }
}
```

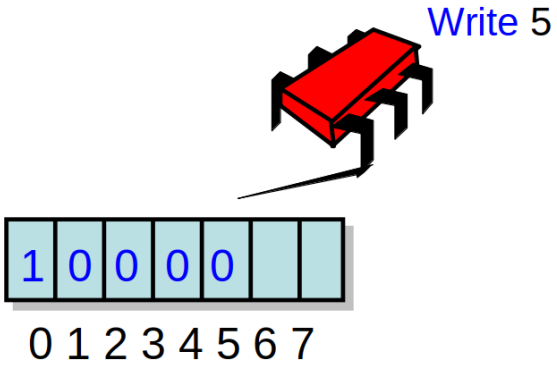
Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean **(3)**
- MRSW Regular **(4)**
- MRSW Atomic
- MRMW Atomic
- Atomic snapshot

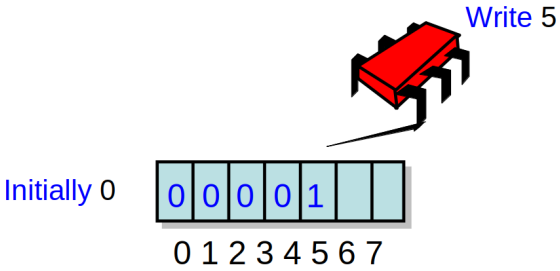
Representing m values



Representing m values



Representing m values



Regular MRSW Integer from Regular MRSW Boolean

```
public class RegIntMRSWRegister implements Register<Integer> {
  RegBoolMRSWRegister[M] bit; // unary representation: bit[i] means value i
  public void write(int x) {
    bit[x].write(true);         // set bit x
    for (int i = x - 1; i >= 0; i--) {
      bit[i].write(false);      // clear bits from higher to lower
    }
  }
  public int read() {
    for (int i = 0; i < M; i++) { // Scan from lower to higher
      if (bit[i].read()) {        // return first bit set
        return i;
      }
    }
  }
```

Critical task №2: easy

Homework, critical

Prove that Regular MRSW Integer construction is correct and wait-free.
Use Section 4.2.3 "A Regular M-Valued MRSW Register"

Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular **(4)**
- MRSW Atomic **(5)**
- MRMW Atomic
- Atomic snapshot

Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular **(4)**
  - SRSW Atomic **(4.5)**
- MRSW Atomic **(5)**
- MRMW Atomic
- Atomic snapshot

Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular **(4)**
  - SRSW Atomic **(4.5)** (proof)
- MRSW Atomic **(5)**
- MRMW Atomic
- Atomic snapshot

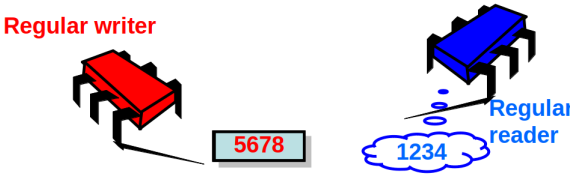
Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular **(4)**
  - SRSW Atomic **(4.5)** (proof)
- MRSW Atomic **(5)** (idea)
- MRMW Atomic
- Atomic snapshot

Register constructions: road map

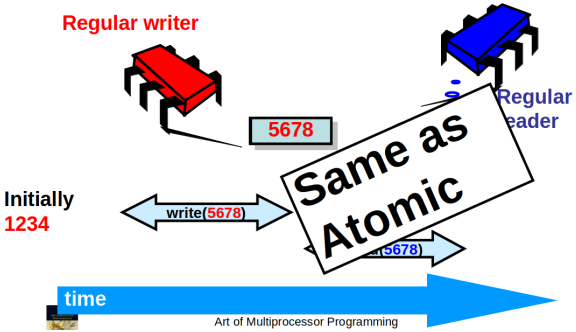
- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular **(4)**
  - SRSW Atomic **(4.5)** (proof)
- MRSW Atomic
- MRMW Atomic
- Atomic snapshot

Atomic SRSW from Regular SRSW: problem

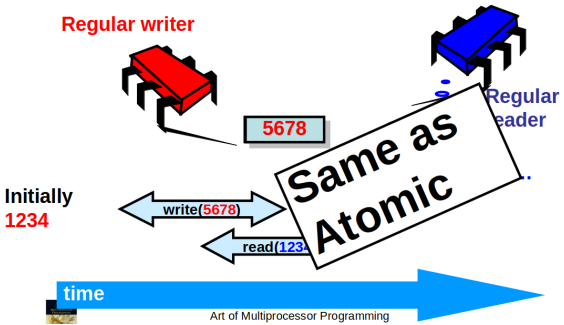


When is this a problem?

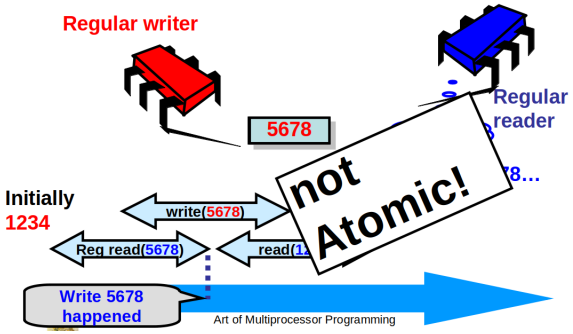
Atomic SRSW from Regular SRSW: problem



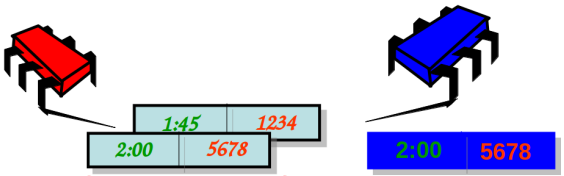
Atomic SRSW from Regular SRSW: problem



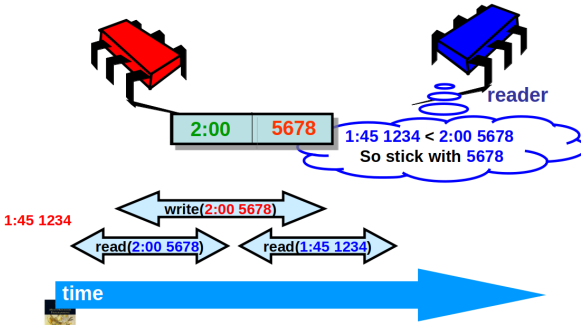
Atomic SRSW from Regular SRSW: problem



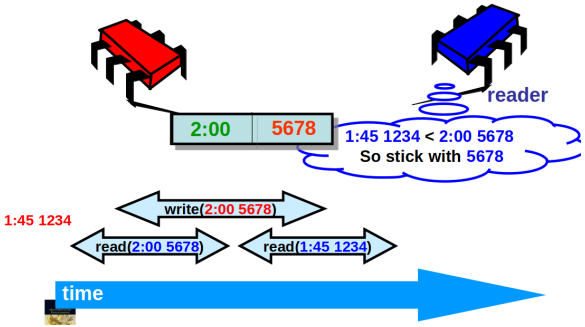
Atomic SRSW from Regular SRSW: timestamped values



Atomic SRSW from Regular SRSW: timestamped values



Atomic SRSW from Regular SRSW: timestamped values



Same as atomic

Timestamped values: clock problem

How different threads coordinate "current time"?

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.
In previous construction we were building **single-writer single-reader** register

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.
In previous construction we were building **single-writer single-reader** register

- Writer was timestamping data

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.
In previous construction we were building **single-writer single-reader** register

- Writer was timestamping data
- Reader used stamps to track "latest" updates

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.
In previous construction we were building **single-writer single-reader** register

- Writer was timestamping data
- Reader used stamps to track "latest" updates

Clock == writer-specific integer counter

Timestamped values: clock problem

How different threads coordinate "current time"?

- They need some kind of consistent global clock
- Mutual exclusion?

We cannot "cheat" by using real clock time.
In previous construction we were building **single-writer single-reader** register

- Writer was timestamping data
- Reader used stamps to track "latest" updates

Clock == writer-specific integer counter
Beware: ensure you understand what is called "timestamp" in the following proofs

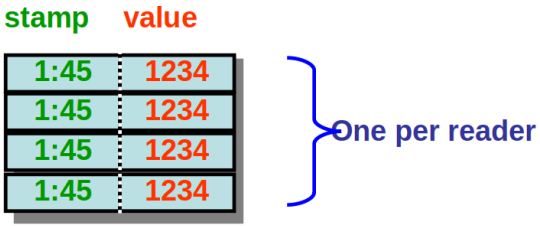
Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular
  - SRSW Atomic **(4.5)**
- MRSW Atomic **(5)**
- MRMW Atomic
- Atomic snapshot

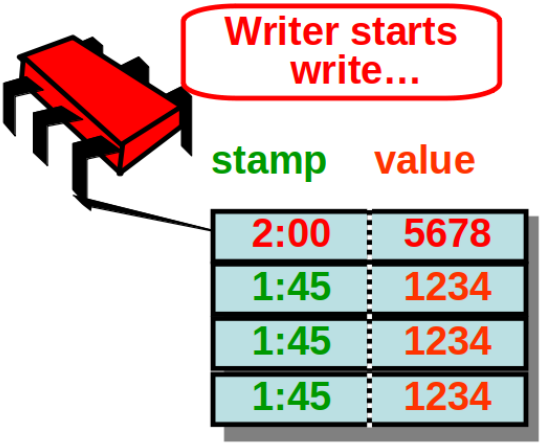
Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular
  - SRSW Atomic **(4.5)**
- MRSW Atomic **(5)** (idea)
- MRMW Atomic
- Atomic snapshot

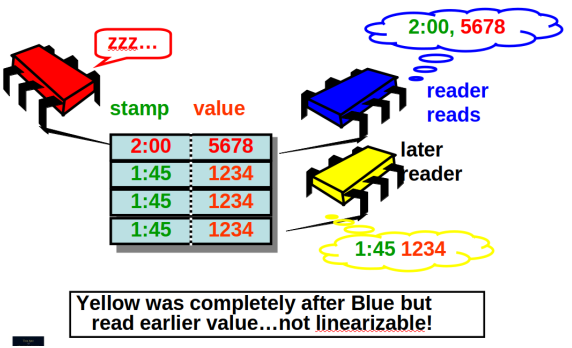
Atomic MRSW from Atomic SRSW: naive idea



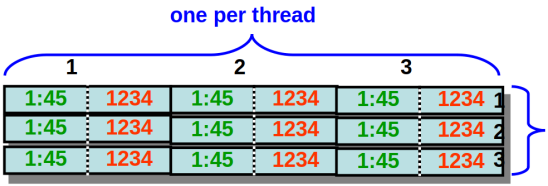
Atomic MRSW from Atomic SRSW: naive idea



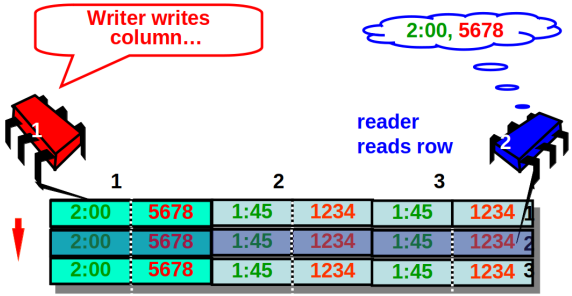
Atomic MRSW from Atomic SRSW: naive idea does not work



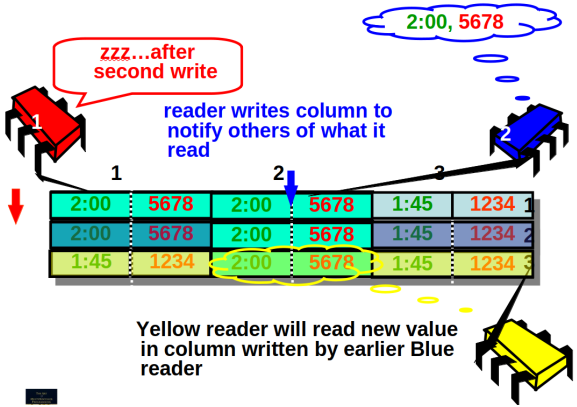
Atomic MRSW from Atomic SRSW: multi-reader table



Atomic MRSW from Atomic SRSW: multi-reader table



Atomic MRSW from Atomic SRSW: multi-reader table



Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable)
- concurrency (multi-reader)
- progress condition (wait-freedom)

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping of values with writer-local clock
- Readers **write** values into auxiliary data structures
- Single high-level operation (atomic read or write) requires N low-level steps (update of thread-specific regular registers)

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping, Readers cooperate with writers, Decomposition

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping, Readers cooperate with writers, Decomposition

which was hard to prove

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping, Readers cooperate with writers, Decomposition

which was hard to prove

- Safe register is unsafe and counter-intuitive
- Regular register oscillate under contention
- Atomic register has "floating" linearization points

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping, Readers cooperate with writers, Decomposition

which was hard to prove

- Safe register is unsafe and counter-intuitive
- Regular register oscillate under contention
- Atomic register has "floating" linearization points

but beneficial

Atomic MRSW from Atomic SRSW: summary

In order to achieve high level of

- consistency (linearizable), concurrency (multi-reader), progress condition (wait-freedom)

we used complicated algorithm with non-obvious design solutions

- Timestamping, Readers cooperate with writers, Decomposition

which was hard to prove

- Safe register is unsafe and counter-intuitive
- Regular register oscillate under contention
- Atomic register has "floating" linearization points

but beneficial

- Any algorithm based on atomic registers could be implemented via safe SRSW boolean registers
- Every register construction is wait-free and does not depend on scheduler

Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular
  - SRSW Atomic
- MRSW Atomic **(5)**
- MRMW Atomic **(6)**
- Atomic snapshot

Register constructions: road map

- SRSW Safe Boolean
- MRSW Safe Boolean
- MRSW Regular Boolean
- MRSW Regular
  - SRSW Atomic
- MRSW Atomic **(5)**
- MRMW Atomic **(6)** (optional homework)
- Atomic snapshot

Atomic MRMW from Atomic MRSW: optional homework

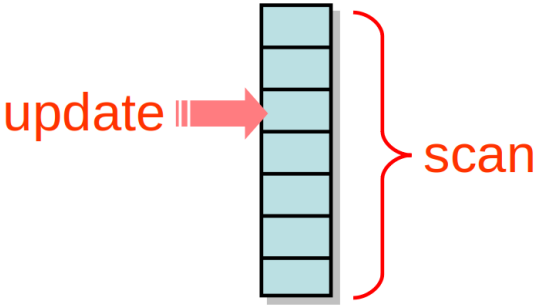
Optional homework: slides 95-103 from `chapter_04.ppt`
Optional homework: Section 4.2.6 "An Atomic MRMW Register" (pages 85-87)
Key insights:

- sort events by lexicographic order and prove it is consistent with *precedence*

Lecture plan

1. Progress conditions
2. Space of registers
3. Register constructions
4. Atomic snapshots
5. Consensus number
6. Summary

Atomic snapshot



Atomic snapshot

Array of MRSW atomic registers (1 register per thread)

- Take instantaneous snapshot of all
- Generalizes to MRMW registers

Atomic snapshot

Array of MRSW atomic registers (1 register per thread)

- Take instantaneous snapshot of all
- Generalizes to MRMW registers

```
interface Snapshot<T> {
  // Thread.currentThread writes `v` to its own register (arr[Thread.id] = v)
  public void update(T v);
  // Instantaneous snapshot of all threads' registers (return copy(arr))
  public T[] scan();
}
```

Atomic snapshot

Array of MRSW atomic registers (1 register per thread)

- Take instantaneous snapshot of all
- Generalizes to MRMW registers

```
interface Snapshot<T> {
  // Thread.currentThread writes `v` to its own register (arr[Thread.id] = v)
  public void update(T v);
  // Instantaneous snapshot of all threads' registers (return copy(arr))
  public T[] scan();
}
```

Collect

- Read values one at a time

Problem

- Incompatible concurrent collects
- Result not linearizable

Clean collect

Clean Collect

- Collect during which nothing changed

Clean collect

Clean Collect

- Collect during which nothing changed

Challenge

- Can we make it happen?
- Can we detect it?

Clean collect

Clean Collect

- Collect during which nothing changed

Challenge

- Can we make it happen?
- Can we detect it?

Simple snapshot:

- Put increasing labels on each entry
- Collect twice
- If both agree – we are done
- Otherwise – try again

Clean collect

Clean Collect

- Collect during which nothing changed

Challenge

- Can we make it happen?
- Can we detect it?

Simple snapshot:

- **Put increasing labels on each entry (why?)**
- Collect twice
- If both agree – we are done
- Otherwise – try again

Clean collect

Clean Collect

- Collect during which nothing changed

Challenge

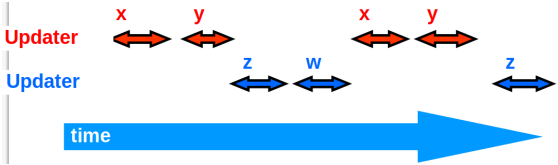
- Can we make it happen?
- Can we detect it?

Simple snapshot:

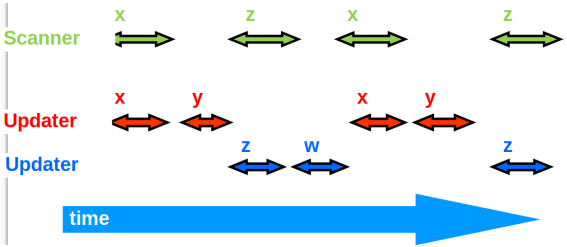
- **Put increasing labels on each entry (why?)**
- Collect twice
- If both agree – we are done
- Otherwise – try again

Problem: Scanner might not be collecting a snapshot!

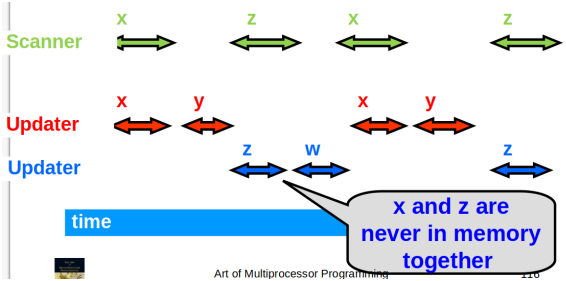
Clean collect: must use labels



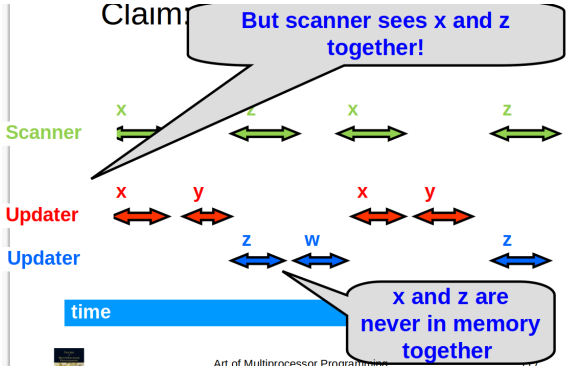
Clean collect: must use labels



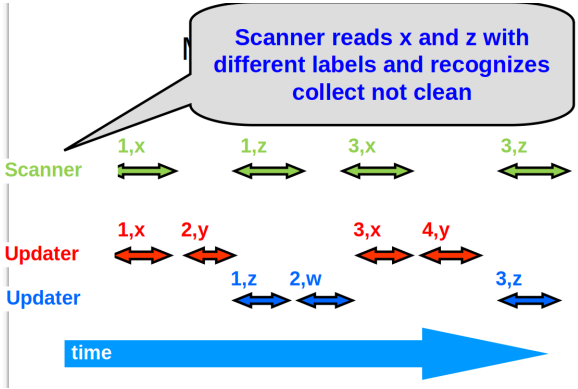
Clean collect: must use labels



Clean collect: must use labels



Clean collect: must use labels



ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.
This is called ABA problem. One of the most complicated issues with non-blocking algorithms.

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.
This is called ABA problem. One of the most complicated issues with non-blocking algorithms. Timestamping helps to solve it.

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.
This is called ABA problem. One of the most complicated issues with non-blocking algorithms. Timestamping helps to solve it. But what if your timestamp (`int` counter) overflow and wrap?

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.
This is called ABA problem. One of the most complicated issues with non-blocking algorithms. Timestamping helps to solve it. But what if your timestamp (`int` counter) overflow and wrap? "The Art of Multiprocessor Programming" section 2.7 "Bounded Timestamps"

ABA: first encounter

We had a subtle concurrent problem:

- Thread 1 reads some data ("current state of system", e.g. `queue.length == 2`), denote **A**
- Thread 2 changes system state to **B** (e.g. `queue.enq`)
- Thread 3 changes system state back to **A** (e.g. `queue.deq`)
- Thread 1 reads system state again

Thread 1 thinks "nothing happened with the system" (e.g. no new data arrived) which is **incorrect**.
This is called ABA problem. One of the most complicated issues with non-blocking algorithms.
Timestamping helps to solve it. But what if your timestamp (`int` counter) overflow and wrap?
"The Art of Multiprocessor Programming" section 2.7 "Bounded Timestamps"
... or just use `long` counters ;)

Simple snapshot

```
private AtomicMRSWRegister[] register; // reg[i] belongs to i-th Thread
public void update(T value) {
  var i = ThreadID.get(); var oldValue = register[i].read();
  register[i].write(new LabeledValue(oldValue.label + 1, value));
}
private LabeledValue[] collect() {
  var copy = new LabeledValue[n];
  for (int j = 0; j < n; j++) copy[j] = register[j].read();
  return copy;
}
public T[] scan(){ var oldCopy = collect();
  while (true) {   var newCopy = collect(); // <-----------------------------+
                   if (equals(oldCopy, newCopy)) return getValues(newCopy);//|
                   oldCopy = newCopy; // -------------------------------------+
  }}
```

Simple snapshot

- Linearizable
- Update is wait-free
  - No unbounded loops
- But Scan can starve
  - If interrupted by concurrent update

Simple snapshot

- Linearizable
- Update is wait-free
  - No unbounded loops
- But Scan can starve (**Obstruction-free**)
  - If interrupted by concurrent update

Critical task №2: medium

Homework, critical

- *Finish "Easy" level.*
- *Prove that "Simple snapshot" algorithm is correct, wait-free for `update()` and obstruction-free for `scan()`. Use Section 4.3.1 "An Obstruction-Free Snapshot".*
- ***Note:** algorithm uses stamped values, be ready to explain how it works in wait-free setting*

Critical task №2: hard

Homework, critical

- *Finish "Medium" level.*
- *Solve exercise 41 from Section 4.5.*

Critical task №2: hard

Exercise 41. There are n processors P_0, ... , P_{n-1} arranged in a ring, where P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link. Each processor keeps a copy of the shared register. To read a register, the processor reads the copy in its local memory.

- A processor P_i starts a `write()` call of value v to register x by sending the message "P_i: `write v to x`" to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_j: `write v to x`" for $i \neq j$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_i: `write v to x`" then it writes v to its local copy of x, and discards the message. The `write()` call is now complete.

Give a short justification or counterexample. If `write()` calls never overlap,

- Is this register implementation regular?
- Is it atomic?

If multiple processors call `write()`,

- Is this register implementation safe?

Atomic snapshot: wait free

We need progress even if other threads update data

Atomic snapshot: wait free

We need progress even if other threads update data

- Every scanner does scan (as before)

Atomic snapshot: wait free

We need progress even if other threads update data

- Every scanner does scan (as before)
- Every updater does scan (*helping hand*)

Atomic snapshot: wait free

We need progress even if other threads update data

- Every scanner does scan (as before)
- Every updater does scan (*helping hand*)
- In case of conflicts scanner could reuse scan from updater

Atomic snapshot: wait free

We need progress even if other threads update data

- Every scanner does scan (as before)
- Every updater does scan (*helping hand*)
- In case of conflicts scanner could reuse scan from updater

Good luck to prove

- Reused value is consistent (resulting snapshot is linearizable)
- Appropriate reused value always exists (wait-freedom for every concurrent scenario)
- Stealing of value is correct

Atomic snapshot: wait free

We need progress even if other threads update data

- Every scanner does scan (as before)
- Every updater does scan (*helping hand*)
- In case of conflicts scanner could reuse scan from updater

Good luck to prove

- Reused value is consistent (resulting snapshot is linearizable)
- Appropriate reused value always exists (wait-freedom for every concurrent scenario)
- Stealing of value is correct

Optional homework: Section 4.3.2 "A Wait-Free Snapshot" and Section 4.3.3 "Correctness Arguments" (pages 88-93)

Register constructions: summary

Safe Single-Reader Single-Writer Boolean register could

- implement other registers, up to Atomic MRMW Integer registers
- provide a consistent atomic snapshot of N registers

using wait-free algorithms

Register constructions: summary

Safe Single-Reader Single-Writer Boolean register could

- implement other registers, up to Atomic MRMW Integer registers
- provide a consistent atomic snapshot of N registers

using wait-free algorithms

What is the next step to attempt with read-write registers?

Register constructions: summary

Safe Single-Reader Single-Writer Boolean register could

- implement other registers, up to Atomic MRMW Integer registers
- provide a consistent atomic snapshot of N registers

using wait-free algorithms

What is the next step to attempt with read-write registers?

- Snapshot means
  - Write any one array element
  - Read multiple array elements

Register constructions: summary

Safe Single-Reader Single-Writer Boolean register could

- implement other registers, up to Atomic MRMW Integer registers
- provide a consistent atomic snapshot of N registers

using wait-free algorithms

What is the next step to attempt with read-write registers?

- Snapshot means
  - Write any one array element
  - Read multiple array elements
- What about atomic writes to multiple locations?

Lecture plan

1. Progress conditions
2. Space of registers
3. Register constructions
4. Atomic snapshots
5. Consensus number
6. Summary

Motivation

Why do we use mutual exclusion or other communication protocol?

Motivation

Why do we use mutual exclusion or other communication protocol?

- For thread coordination

Motivation

Why do we use mutual exclusion or other communication protocol?

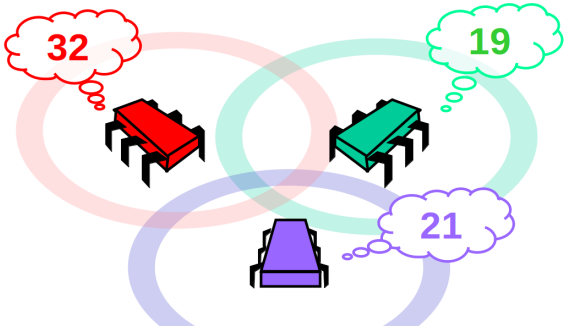
- For thread coordination
- Ensure system evolves according to some rules

Motivation

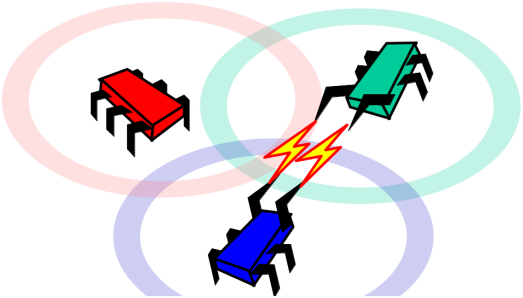
Why do we use mutual exclusion or other communication protocol?

- For thread coordination
- Ensure system evolves according to some rules
- Threads decide what to do together

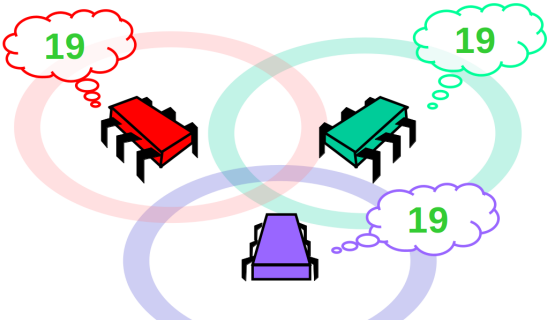
Each thread has private input



They communicate



They agree on some input



Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Read-write registers formalized in terms of safe/regular/atomic concurrent objects.

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Read-write registers formalized in terms of safe/regular/atomic concurrent objects.
Theorem could be adapted to:

- Registers
- Message-passing
- Carrier pigeons
- Any kind of asynchronous computation

Consensus: why?

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Consensus: why?

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Theorem helps to prove fun stuff, e.g.

- it is impossible to implement a two-dequeuer wait-free FIFO queue
- from read/write memory

Consensus: why?

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Theorem helps to prove fun stuff, e.g.

- it is impossible to implement a two-dequeuer wait-free FIFO queue
- from read/write memory

Let's classify all concurrent objects by their "synchronization power"

Consensus number

An object **X** has *consensus number* **n**

- If it can be used to solve **n**-thread consensus
  - Take any number of instances of **X**
  - together with atomic read/write registers
  - and implement **n**-thread consensus
- But not **(n+1)**-thread consensus

Consensus number

An object **X** has *consensus number* **n**

- If it can be used to solve **n**-thread consensus
  - Take any number of instances of **X**
  - together with atomic read/write registers
  - and implement **n**-thread consensus
- But not **(n+1)**-thread consensus

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Consensus number

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Consensus number

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement **wait-free** multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Consensus number

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement **wait-free** multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Theorem

Multi-dequeuer FIFO queues have consensus number at least 2

Atomic registers cannot implement highly concurrent wait-free FIFO

- it is impossible to implement a two-dequeuer wait-free FIFO queue
- from read/write memory

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Registers have consensus number 1

Consensus numbers measure synchronization power

Theorem

- *If you can implement* ***X*** *from* ***Y***
- *And* ***X*** *has consensus number* ***c***
- *Then* ***Y*** *has consensus number at least* ***c***

Registers have consensus number **1**

There are practically interesting problems that require higher consensus number

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Registers have consensus number **1**
There are practically interesting problems that require higher consensus number
What should we do next?

Summary

Progress conditions

- Dependent progress: Deadlock-freedom, Starvation-freedom
- Non-blocking progress: Lock-freedom, Wait-freedom
- Dependent non-blocking progress: Obstruction-freedom

Register design space

- Capacity, Concurrency (SRSW, MRMW, MRMR), Consistency (safe, regular, atomic)
- Atomic snapshot

Wait-free constructions using

- Timestamps, Helping/Cooperation, Decomposition

New problems specific to non-blocking designs

- Global clock, ABA

Consensus number as a tool to formalize relative synchronization power

Summary: homework

Homework, critical

Easy. *Prove that Regular MRSW Integer construction is correct and wait-free. Use Section 4.2.3 "A Regular M-Valued MRSW Register"*

Homework, critical

Medium.

- *Prove that "Simple snapshot" algorithm is correct, wait-free for* `update()` *and obstruction-free for* `scan()`*. Use Section 4.3.1 "An Obstruction-Free Snapshot".*
- ***Note:*** *algorithm uses stamped values, be ready to explain how it works in wait-free setting*

Homework, critical

Hard. *Solve exercise 41 from Section 4.5.*