Lecture 6: theoretical foundations of concurrency

timeline, events, precedence, 2-thread mutual exclusion, deadlock freedom, starvation freedom, N-thread mutual exclusion, sequential objects and specifications, concurrent objects, linearizability

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l6.pdf

In previous episodes

Concurrency is complicated due to:

- Independence of agents
  - combinatorial explosion of possible executions
- Uncontrollable execution speed of agents
  - spurious failures
- Uncertainty with definition of consistency
  - concurrent invariants
  - visibility
  - data race
  - atomicity

In previous episodes

Concurrency is complicated due to:

- Independence of agents
  - combinatorial explosion of possible executions
- Uncontrollable execution speed of agents
  - spurious failures
- Uncertainty with definition of consistency
  - concurrent invariants
  - visibility
  - data race
  - atomicity

Is it purely engineering issue or we have encountered some fundamental problem?

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

It usually takes several mathematically-intensive sessions to go from the definition to really interesting theorems.

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

It usually takes several mathematically-intensive sessions to go from the definition to really interesting theorems.
Concurrency is more "inherently-complicated" so it takes even more time to rigorously prove non-trivial properties.

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

It usually takes several mathematically-intensive sessions to go from the definition to really interesting theorems.
Concurrency is more "inherently-complicated" so it takes even more time to rigorously prove non-trivial properties.
We will skip or simplify some steps.

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

It usually takes several mathematically-intensive sessions to go from the definition to really interesting theorems.
Concurrency is more "inherently-complicated" so it takes even more time to rigorously prove non-trivial properties.
We will skip or simplify some steps.
There still will be a lot of technical (yet mathematically heavy) parts.

Be patient

Reminder: you are studying Turing Machines or Finite-State Machines in order to

- get a compact and fully formalized description of some "computing device"
- understand theoretical limitations of an object (Halting Problem, Pumping Lemma)

It usually takes several mathematically-intensive sessions to go from the definition to really interesting theorems.
Concurrency is more "inherently-complicated" so it takes even more time to rigorously prove non-trivial properties.
We will skip or simplify some steps.
There still will be a lot of technical (yet mathematically heavy) parts.
At least, you will understand why proving concurrent algorithms could be really hard.

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_02.ppt`
- `chapter_03.ppt`

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.
It looks like Java and feels like Java but algorithms from the slides **will not work** as expected if run on the JVM.

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.
It looks like Java and feels like Java but algorithms from the slides **will not work** as expected if run on the JVM.
In Lecture 9 and Lecture 10 we will understand why.

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.
It looks like Java and feels like Java but algorithms from the slides **will not work** as expected if run on the JVM.
In Lecture 9 and Lecture 10 we will understand why.
We are going to do some real Computer Science, it should work from mathematical point of view ;)

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.
It looks like Java and feels like Java but algorithms from the slides **will not work** as expected if run on the JVM.
In Lecture 9 and Lecture 10 we will understand why.
We are going to do some real Computer Science, it should work from mathematical point of view ;)
Theoretical intuition and insights will help us to understand how to implement efficient and correct algorithms using existing hardware.

Theoretical mode enabled

Important: in Lecture 6 (today) and Lecture 7 (next time) we will use **pseudo**code.
It looks like Java and feels like Java but algorithms from the slides **will not work** as expected if run on the JVM.
In Lecture 9 and Lecture 10 we will understand why.
We are going to do some real Computer Science, it should work from mathematical point of view ;)
Theoretical intuition and insights will help us to understand how to implement efficient and correct algorithms using existing hardware.
Hint: focus on ideas rather mathematical formalisms.

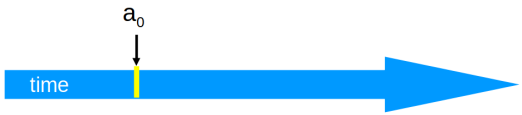
Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Time and events

An **event** a_0 of thread `A` is

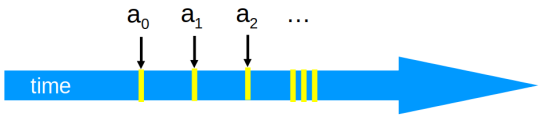
- Instantaneous
- No simultaneous events (break ties)



Threads

A thread A is (formally) a sequence a_0, a_1, ... of events

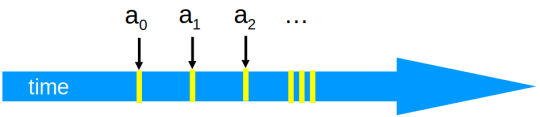
- "Trace" model
- Notation: $a_0 \rightarrow a_1$ indicates order



Threads

A thread A is (formally) a sequence a_0, a_1, ... of events

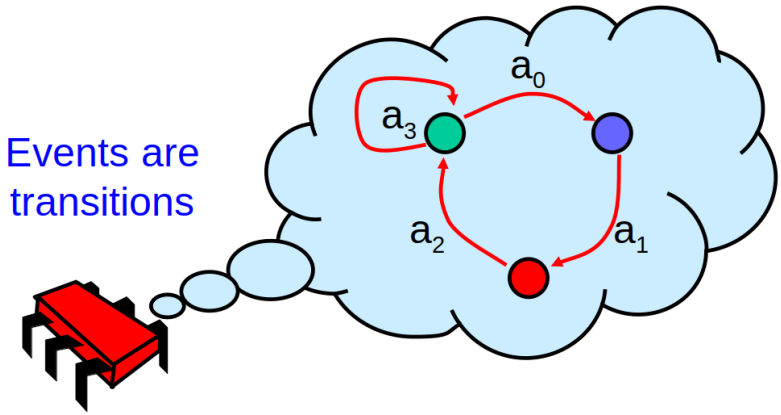
- "Trace" model
- Notation: $a_0 \rightarrow a_1$ indicates order



Example Thread events:

- Assign to shared or local variable
- Invoke method
- Return from method
- ...

Threads are state machines



Concurrent threads

Thread A

Thread B

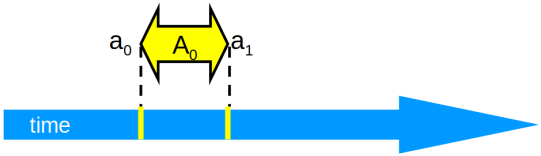
Events of two or more threads:

- Interleaved
- Not necessarily independent

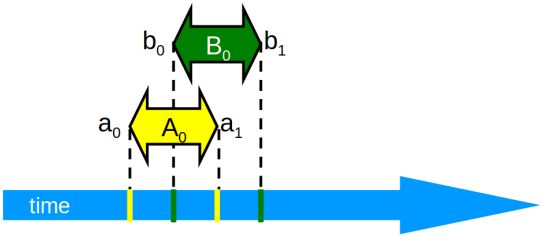
Methods are intervals

An **interval** $A_0 = (a_0, a_1)$ is

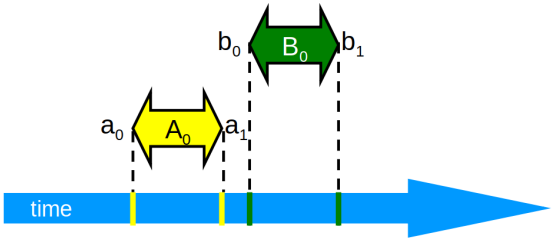
- Time between events a_0 and a_1



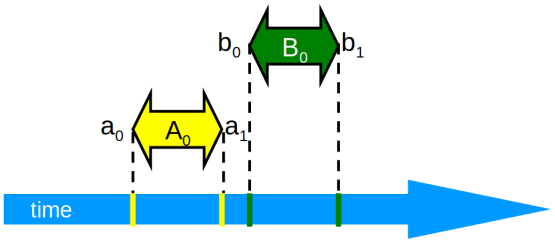
Overlapping intervals



Disjoint intervals



Disjoint intervals



Precedence:

- Interval A_0 **precedes** interval B_0
- Notation: $A_0 \rightarrow B_0$

Precedence

$A_0 \rightarrow B_0$

- A_0 **precedes** interval B_0

Precedence

$A_0 \rightarrow B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0

Precedence

$A_0 \rightarrow B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\rightarrow$ Renaissance"

Precedence

$A_0 \rightarrow B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\rightarrow$ Renaissance"

what about *this week* vs *this month*?

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?

:

- Never true that $A \to A$.
- If $A \to B$ then not true $B \to A$.
- If $A \to B$ and $B \to C$ then $A \to C$.

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?

:

- Never true that $A \to A$.
- If $A \to B$ then not true $B \to A$.
- If $A \to B$ and $B \to C$ then $A \to C$.
- Remember: $A \to B$ and $B \to A$ might both be false!

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?
Precedence is a partial order:

- Never true that $A \to A$.
- If $A \to B$ then not true $B \to A$.
- If $A \to B$ and $B \to C$ then $A \to C$.
- Remember: $A \to B$ and $B \to A$ might both be false!

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?
Precedence is a partial order:

- Never true that $A \to A$. (Irreflexive)
- If $A \to B$ then not true $B \to A$.
- If $A \to B$ and $B \to C$ then $A \to C$.
- Remember: $A \to B$ and $B \to A$ might both be false!

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?
Precedence is a partial order:

- Never true that $A \to A$. (Irreflexive)
- If $A \to B$ then not true $B \to A$. (Antisymmetric)
- If $A \to B$ and $B \to C$ then $A \to C$.
- Remember: $A \to B$ and $B \to A$ might both be false!

Precedence

$A_0 \to B_0$

- A_0 **precedes** interval B_0
- A_0 **happens before** interval B_0
- Fancy way to say "Middle Ages $\to$ Renaissance"

what about *this week* vs *this month*?
Precedence is a partial order:

- Never true that $A \to A$. (Irreflexive)
- If $A \to B$ then not true $B \to A$. (Antisymmetric)
- If $A \to B$ and $B \to C$ then $A \to C$. (Transitive)
- Remember: $A \to B$ and $B \to A$ might both be false!

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Mutual exclusion

Thread is *well formed* if:

- each critical section is associated with a unique `Lock` object
- `Lock.lock` when trying to enter critical section
- `Lock.unlock` when leaving critical section

Let CS_A^j be the interval: thread `A` executes critical section for `j`-th time.

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.
Mutual exclusion:

- Critical sections of different threads do not overlap.

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.

Mutual exclusion:

- Critical sections of different threads do not overlap. $\forall A, B, k, j \; CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.

Mutual exclusion:

- Critical sections of different threads do not overlap. $\forall A, B, k, j \; CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$

Freedom from deadlock:

- If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread `A` calls `lock()` but never acquires the lock, then other threads must be completing an infinite number of critical sections.

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.

Mutual exclusion:

- Critical sections of different threads do not overlap. $\forall A, B, k, j \; CS_A^k \rightarrow CS_B^j$ or $CS_B^j \rightarrow CS_A^k$

Freedom from deadlock:

- If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread `A` calls `lock()` but never acquires the lock, then other threads must be completing an infinite number of critical sections.

Freedom from starvation (lockout freedom):

- Every thread that attempts to acquire the lock eventually succeeds. Every call to `lock()` eventually returns.

Mutual exclusion for 2 agents

CS_A^j: thread `A` executes critical section for `j`-th time.

Mutual exclusion:

- Critical sections of different threads do not overlap. $\forall A, B, k, j \; CS_A^k \to CS_B^j$ or $CS_B^j \to CS_A^k$

Freedom from deadlock:

- If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread `A` calls `lock()` but never acquires the lock, then other threads must be completing an infinite number of critical sections.

Freedom from starvation (lockout freedom):

- Every thread that attempts to acquire the lock eventually succeeds. Every call to `lock()` eventually returns.

Theorem

Starvation freedom implies deadlock freedom.

Question time

Question: Mutual exclusion, Freedom from deadlock, Freedom from starvation. Which are safety properties, which are liveness properties?

Question time

Question: Restaurant "Concurrency pitfalls" claims that every philosopher will eat ordered meal before Sun explodes. Does it provide **Freedom from starvation**?

LockOne

```
class LockOne implements Lock {
  boolean[] flag = new boolean[2];
  void lock() {
    int i = ThreadID.get(); // 0 or 1
    int j = 1 - i;
    flag[i] = true;         // I want to enter
    while (flag[j]) {}      // await other thread
  }
  void unlock() {
    int i = ThreadID.get();
    flag[i] = false;        // I do not want to enter any more
  }
}
```

LockOne: deadlock

```
void lock() {
  int i = ThreadID.get();
  int j = 1 - i;
  flag[i] = true;
  while (flag[j]) {}
}
void unlock() {
  flag[ThreadID.get()] = false;
}
```

$write_A(flag[A] = true)$ occur before $read_B(flag[A])$
$write_B(flag[B] = true)$ occur before $read_A(flag[B])$

LockOne: mutual exclusion

```
void lock() {
  flag[i] = true;
  while (flag[j]) {}
}
void unlock() {
  flag[i] = false;
}
```

From the code:

- $write_A(flag[A] = true) \rightarrow read_A(flag[B] = false) \rightarrow CS_A$
- $write_B(flag[B] = true) \rightarrow read_B(flag[A] = false) \rightarrow CS_B$

LockOne: mutual exclusion

Plan:

- Assume CS_A^j overlaps CS_B^k.
- Consider each thread's last (j-th and k-th) read and write in the `lock()` method before entering
- Derive a contradiction

LockOne: mutual exclusion

Plan:

- Assume CS_A^j overlaps CS_B^k.
- Consider each thread's last (j-th and k-th) read and write in the `lock()` method before entering
- Derive a contradiction

Informal hint: Thread A observed `flag[B] = false` and entered CS_A^j **and then** Thread B entered CS_B^k which required to write `flag[B] = true`

LockOne: mutual exclusion

Plan:

- Assume CS_A^j overlaps CS_B^k.
- Consider each thread's last (j-th and k-th) read and write in the `lock()` method before entering
- Derive a contradiction

Informal hint: Thread A observed `flag[B] = false` and entered CS_A^j **and then** Thread B entered CS_B^k which required to write `flag[B] = true`
From assumption:

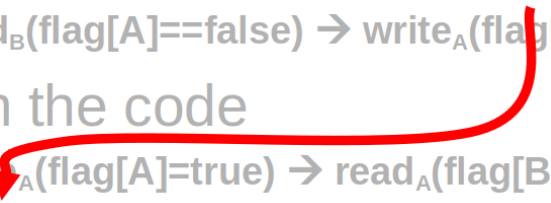
- $read_A(flag[B] = false) \rightarrow write_B(flag[B] = true)$
- $read_B(flag[A] = false) \rightarrow write_A(flag[A] = true)$

LockOne: mutual exclusion

- Assumptions:
  - $\mathbf{read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)}$
  - $\mathbf{read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)}$
- From the code
  - $\mathbf{write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)}$
  - $\mathbf{write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)}$

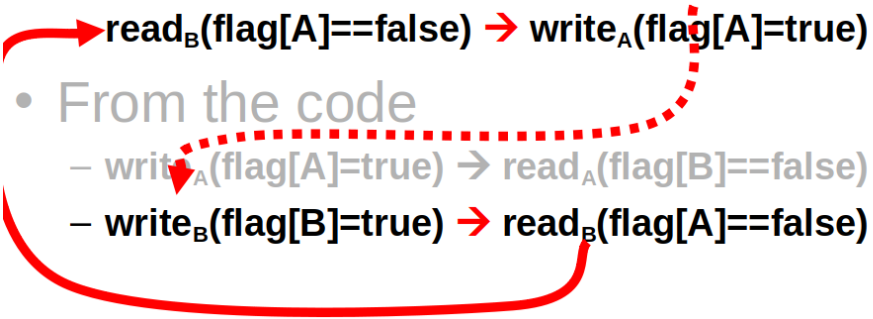
LockOne: mutual exclusion

- Assumptions:
  - $\mathbf{read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)}$
  - $\mathbf{read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)}$
- From the code
  - $\mathbf{write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)}$
  - $\mathbf{write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)}$



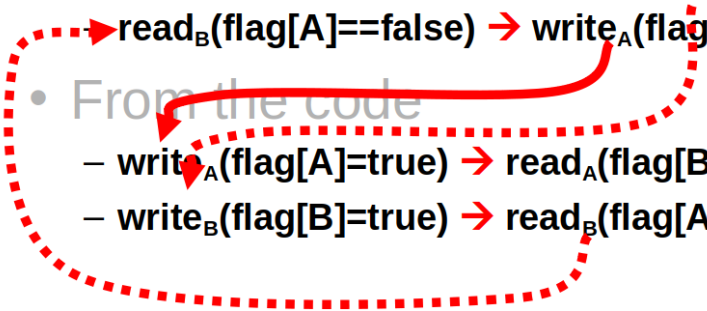
LockOne: mutual exclusion

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$



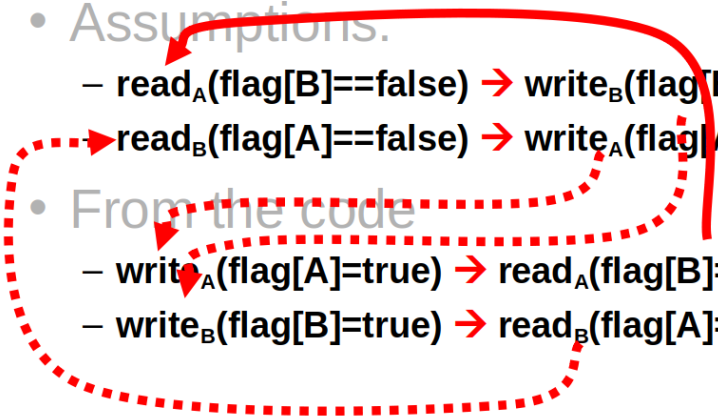
LockOne: mutual exclusion

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$



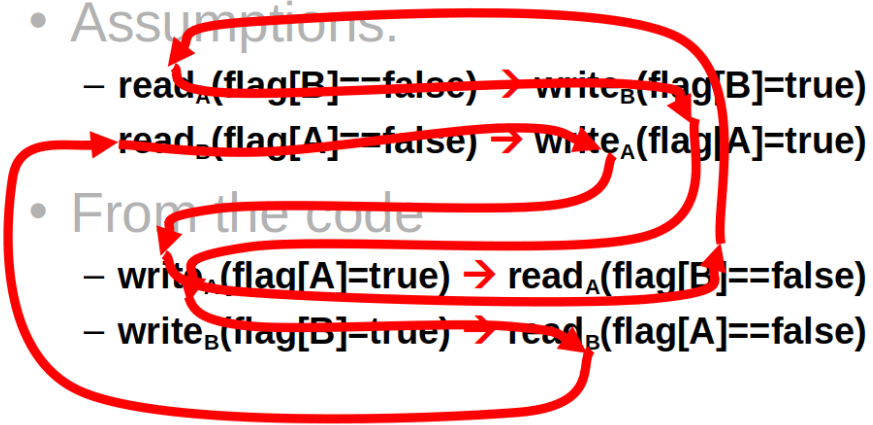
LockOne: mutual exclusion

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

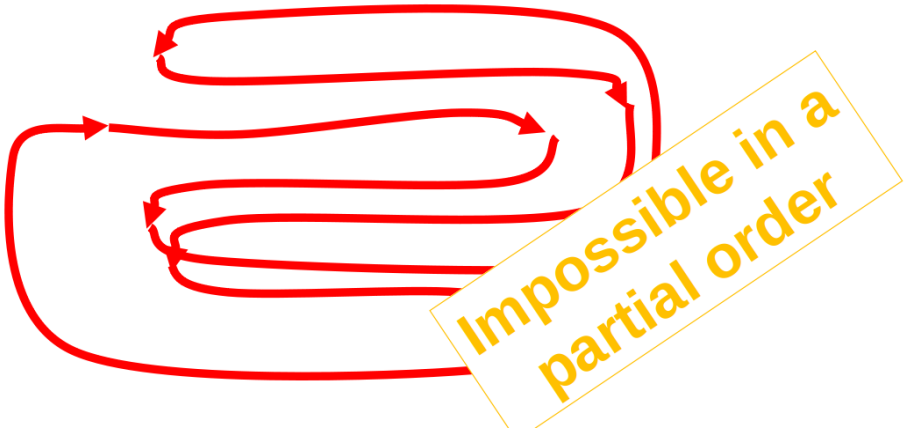


LockOne: mutual exclusion

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$



LockOne: mutual exclusion



LockOne

Properties

- Mutual exclusion
- Deadlock-freedom

LockOne

Properties

- Mutual exclusion
- Deadlock-freedom

```
flag[i] = true;
while (flag[j]) {}
```

```
flag[j] = true;
while (flag[i]) {}
```

LockOne

Properties

- Mutual exclusion
- Deadlock-freedom

```
flag[i] = true;                flag[j] = true;
while (flag[j]) {}             while (flag[i]) {}
```

- Concurrent executions can deadlock
- Sequential executions OK

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  int i = ThreadID.get(); // 0 or 1
  victim = i;             // let the other go first
  while (victim == i) {}; // wait
 }
 public void unlock() {}
}
```

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  int i = ThreadID.get(); // 0 or 1
  victim = i;             // let the other go first
  while (victim == i) {}; // wait
 }
 public void unlock() {}
}
```

Satisfies Mutual Exclusion:

- if thread `i` in *CS*
- then `victim == j`
- cannot be both 0 and 1

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  int i = ThreadID.get(); // 0 or 1
  victim = i;             // let the other go first
  while (victim == i) {}; // wait
 }
 public void unlock() {}
}
```

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  int i = ThreadID.get(); // 0 or 1
  victim = i;             // let the other go first
  while (victim == i) {}; // wait
 }
 public void unlock() {}
}
```

Not deadlock free:

- Sequential execution deadlocks
- Concurrent execution does not

LockOne and LockTwo

LockOne, LockTwo:

- Mutual exclusion
- Deadlock-freedom

LockOne and LockTwo

LockOne, LockTwo:

- Mutual exclusion
- Deadlock-freedom

LockOne:

- Concurrent executions can deadlock
- Sequential executions OK

LockOne and LockTwo

LockOne, LockTwo:

- Mutual exclusion
- Deadlock-freedom

LockOne:

- Concurrent executions can deadlock
- Sequential executions OK

LockTwo:

- Concurrent execution OK
- Sequential execution can deadlock

LockOne and LockTwo

LockOne, LockTwo:

- Mutual exclusion
- Deadlock-freedom

LockOne:

- Concurrent executions can deadlock
- Sequential executions OK

LockTwo:

- Concurrent execution OK
- Sequential execution can deadlock

Idea: combine them!

Peterson's Algorithm

```
class Peterson implements Lock {
  private boolean[] flag = new boolean[2];
  private int victim;
  public void lock() {
    int i = ThreadID.get(); // 0 or 1
    int j = 1 - i;
    flag[i] = true;          // I am interested
    victim  = i;             // you go first
    while (flag[j] && victim == i) {}; // wait while other interested and
                                       // I am the victim
  }
  public void unlock() {
   flag[ThreadID.get()] = false; // I am not interested
  }
}
```

Peterson's Algorithm

Mutual exclusion

```
public void lock() {
  flag[i] = true;          // I am interested
  victim  = i;             // you go first
  while (flag[j] && victim == i) {}; // wait while other interested and
                                     // I am the victim
}
```

From the code:

- $write_B(flag[B] = true) \rightarrow write_B(victim = B)$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Assumption (both threads entered CS, latest to write victim is thread `A`):

- $write_B(victim = B) \rightarrow write_A(victim = A)$

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow write_B(victim = B)$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$
- $write_B(victim = B) \rightarrow write_A(victim = A)$

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow write_B(victim = B)$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$
- $write_B(victim = B) \rightarrow write_A(victim = A)$

Let's swap second and third lines.

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow write_B(victim = B)$
- $write_B(victim = B) \rightarrow write_A(victim = A)$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow write_B(victim = B)$
- $write_B(victim = B) \rightarrow write_A(victim = A)$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Let's remove repeats.

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow$
- $write_B(victim = B) \rightarrow$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow$
- $write_B(victim = B) \rightarrow$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Thread `A` read `flag[B] == true` and `victim == A` so it could not have entered the CS.

Peterson's Algorithm

Mutual exclusion

- $write_B(flag[B] = true) \rightarrow$
- $write_B(victim = B) \rightarrow$
- $write_A(victim = A) \rightarrow read_A(flag[B]) \rightarrow read_A(victim)$

Thread `A` read `flag[B] == true` and `victim == A` so it could not have entered the *CS*. QED.

Peterson's Algorithm

Starvation freedom (implies deadlock freedom)

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

Peterson's Algorithm

Starvation freedom (implies deadlock freedom)

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

- Thread `i` blocked only if `j` repeatedly re-enters so `flag[j] == true` and `victim == i`.

Peterson's Algorithm

Starvation freedom (implies deadlock freedom)

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

- Thread `i` blocked only if `j` repeatedly re-enters so `flag[j] == true` and `victim == i`.
- When thread `j` re-enters it sets `victim = j`.

Peterson's Algorithm

Starvation freedom (implies deadlock freedom)

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
}
public void unlock() {
  flag[i] = false;
}
```

- Thread `i` blocked only if `j` repeatedly re-enters so `flag[j] == true` and `victim == i`.
- When thread `j` re-enters it sets `victim = j`.
- So thread `i` gets in.

Summary: 2-thread mutual exclusion

Formalize concurrent method execution:

- Single timeline
- Totally ordered atomic events
- Methods represented as partially ordered intervals
- *Precedence* is irreflexive, antisymmetric, transitive partial order

Summary: 2-thread mutual exclusion

Formalize concurrent method execution:

- Single timeline
- Totally ordered atomic events
- Methods represented as partially ordered intervals
- *Precedence* is irreflexive, antisymmetric, transitive partial order

Formalize mutex properties:

- Mutual exclusion
- Deadlock freedom
- Starvation freedom (implies Deadlock freedom)

Summary: 2-thread mutual exclusion

Formalize concurrent method execution:

- Single timeline
- Totally ordered atomic events
- Methods represented as partially ordered intervals
- *Precedence* is irreflexive, antisymmetric, transitive partial order

Formalize mutex properties:

- Mutual exclusion
- Deadlock freedom
- Starvation freedom (implies Deadlock freedom)

Prove theorems:

- Algorithm has property (*suppose not ...*)
- Algorithm lacks property (provide counterexample)

Lecture plan

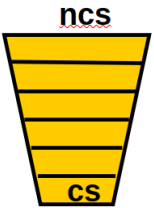
1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Filter Lock

Idea

There are n-1 “waiting rooms” called levels.

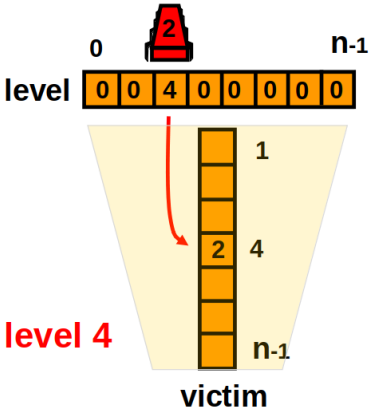
- At each level
  - At least one enters level
  - At least one blocked if many try
- Only one thread makes it through



Filter Lock

Level and victim

```
class Filter implements Lock {
  // level[i] for thread i
  int[] level;
  // victim[L] for level L
  int[] victim;
  public Filter(int n) {
    level  = new int[n];
    victim = new int[n];
    for (int i = 1; i < n; i++) {
      level[i] = 0;
    }
  }
}
```



Thread 2 at level 4

Filter Lock

```
class Filter implements Lock {
  …

  public void lock(){
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃ k != i level[k] >= L) &&
             victim[L] == i ) {};
    }}
  public void unlock() {
    level[i] = 0;
  }}
```

Filter Lock

```
class Filter implements Lock {
  …

  public void lock() {
    for (int L = 1; L < n; L++) {
      level[i] = L;
      victim[L] = i;
      while ((∃ k != i) level[k] >= L) &&
            victim[L] == i) {};
    }}
  public void release(int i) {
    level[i] = 0;
  }}
```

One level at a time

Filter Lock

```
class Filter implements Lock {
  …

  public void lock() {
    for (int L = 1; L < n; L++) {
      level[i] = L;
      victim[L] = i;
      while ((∃ k != i) level[k] >= L) &&
             victim[L] == i)
    }}
  public void release(int i)
    level[i] = 0;
  }}
```

Announce intention to enter level L

Filter Lock

```
class Filter implements Lock {
  int level[n];
  int victim[n];
  public void lock() {
    for (int L = 1; L < n; L++) {
      level[i] = L;
      victim[L] = i;
      while ((∃ k != i) level[k] >= L) &&
            victim[L] == i) {};
    }}
  public void release(int i)
    level[i] = 0;
  }}
```

Give priority to anyone but me

Filter Lock

Wait as long as someone else is at same or higher level, and I'm designated victim

```
public void lock() {
  for (int L = 1; L < n; L++) {
    level[i] = L;
    victim[L] = i;
    while ((∃ k != i) level[k] >= L) &&
          victim[L] == i) {};
  }}
public void release(int i) {
  level[i] = 0;
}}
```

Filter Lock

```
class Filter implements Lock {
  int level[n];
  int victim[n];
  public void lock() {
    for (int L = 1; L < n; L++) {
      level[i]  = L;
      victim[L] = i;
      while ((∃ k != i) level[k] >= L) &&
            victim[L] == i) {};
    }}
```

Thread *enters* level L when it completes the loop

Filter Lock

Proof idea

- Start at level `L = 0`
- At most `n - L` threads enter level `L`
- Mutual exclusion at level `L = n-1`

Filter Lock

Proof idea

- Start at level `L = 0`
- At most `n - L` threads enter level `L`
- Mutual exclusion at level `L = n-1`

Induction Hypothesis:

- No more than `n-(L-1)` at level `L-1`
- Induction step: by contradiction
- Assume all at level `L-1` enter level `L`
- Thread `A` last to write `victim[L]`
- Thread `B` is any other thread at level `L`

Filter Lock

Proof idea

- Start at level `L = 0`
- At most `n - L` threads enter level `L`
- Mutual exclusion at level `L = n-1`

Induction Hypothesis:

- No more than `n-(L-1)` at level `L-1`
- Induction step: by contradiction
- Assume all at level `L-1` enter level `L`
- Thread `A` last to write `victim[L]`
- Thread `B` is any other thread at level `L`

Homework: slides 77-93 from `chapter_02.ppt`

Homework: Section 2.4 "The Filter Lock" (pages 28 - 31) from "The Art of Multiprocessor Programming"

Filter Lock

No starvation

Filter Lock satisfies properties:

- Just like Peterson Alg at any level
- So no one starves

Filter Lock

No starvation

Filter Lock satisfies properties:

- Just like Peterson Alg at any level
- So no one starves

But what about fairness?

Filter Lock

No starvation

Filter Lock satisfies properties:

- Just like Peterson Alg at any level
- So no one starves

But what about fairness?
Threads can be overtaken by others

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?
- **What** are those registers? How to describe them formally?

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?
- **What** are those registers? How to describe them formally?
- Is there any **alternative** to registers and how to use it to implement mutual exclusion?

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?
  Theorem about Lower bounds, next slide
- **What** are those registers? How to describe them formally?
- Is there any **alternative** to registers and how to use it to implement mutual exclusion?

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?
  Theorem about Lower bounds, next slide
- **What** are those registers? How to describe them formally?
  Concurrent objects chapter, later today
- Is there any **alternative** to registers and how to use it to implement mutual exclusion?

[1] Historical name

Summary: N-thread mutual exclusion

N-level design:

- every step "filters"(prevents from progress) at least one thread
- reuse ideas from simpler design (2-thread Peterson's algorithm)

We have many memory cells (**registers**[1]): boolean `flag[i]`, integer `victim[j]`

- **How many** registers are needed to solve N-thread mutual exclusion?
  Theorem about Lower bounds, next slide
- **What** are those registers? How to describe them formally?
  Concurrent objects chapter, later today
- Is there any **alternative** to registers and how to use it to implement mutual exclusion?
  Lecture 8

[1] Historical name

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Mutual exclusion for N threads requires read-write of at least N locations

Optional homework:

> *Theorem 2.8.1. Any* `Lock` *algorithm that, by reading and writing memory, solves deadlock-free mutual exclusion for three threads must use at least three distinct memory locations.*

Key insights:

- Covering state

Generalized theorem: N-thread deadlock-free mutual exclusion requires N distinct locations.

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For `N` concurrent threads
- Using `N` boolean and `N` integer registers
- Provides starvation-freedom

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For `N` concurrent threads
- Using `N` boolean and `N` integer registers
- Provides starvation-freedom

Mutual exclusion could not be solved:

- Using less than `N` registers

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For N concurrent threads
- Using N boolean and N integer registers
- Provides starvation-freedom

Mutual exclusion could not be solved:

- Using less than N registers

Few questions remain:

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For `N` concurrent threads
- Using `N` boolean and `N` integer registers
- Provides starvation-freedom

Mutual exclusion could not be solved:

- Using less than `N` registers

Few questions remain:

- How much time will thread wait until success? `FilterLock` does not guarantee fairness

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For `N` concurrent threads
- Using `N` boolean and `N` integer registers
- Provides starvation-freedom

Mutual exclusion could not be solved:

- Using less than `N` registers

Few questions remain:

- How much time will thread wait until success? `FilterLock` does not guarantee fairness
- Could we "improve" registers with some additional operations to speed-up the algorithm?

Mutual exclusion: you know so far

Mutual exclusion could be solved:

- For `N` concurrent threads
- Using `N` boolean and `N` integer registers
- Provides starvation-freedom

Mutual exclusion could not be solved:

- Using less than `N` registers

Few questions remain:

- How much time will thread wait until success? `FilterLock` does not guarantee fairness
- Could we "improve" registers with some additional operations to speed-up the algorithm?
- What if threads are created and deleted dynamically?

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Formalizing fairness

Optional homework: slides 95-99 from `chapter_02.ppt`
Optional homework: Section 2.5 "Fairness" (page 31)
Key insights:

- *doorway* section
- *waiting* section
- first-come-first-served ($D_A^j \rightarrow D_B^k$ then $CS_A^j \rightarrow CS_B^k$)

Bakery algorithm

Optional homework: slides 99-118 from `chapter_02.ppt`
Optional homework: Section 2.6 "Lamport's Bakery Algorithm" (pages 31 - 33)
Key insights:

- monotonically growing labels for operations
- proving fairness

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

We could classify registers by allowed concurrency:

- Single Reader Single Writer (SRSW)
- Multiple Reader Single Writer (MRSW)
- Multiple Reader Multiple Writer (MRMW)

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

We could classify registers by allowed concurrency:

- Single Reader Single Writer (SRSW)
- Multiple Reader Single Writer (MRSW)
- Multiple Reader Multiple Writer (MRMW)

We could classify registers by capacity:

- Boolean register (*true*/*false*), Integer register $(0, 1, ..., 2^N)$

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

We could classify registers by allowed concurrency:

- Single Reader Single Writer (SRSW)
- Multiple Reader Single Writer (MRSW)
- Multiple Reader Multiple Writer (MRMW)

We could classify registers by capacity:

- Boolean register (*true*/*false*), Integer register $(0, 1, ..., 2^N)$

We could classify registers by supported operations:

- Read-only, Write-once, Monotonically increasing, Arbitrary update

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

We could classify registers by allowed concurrency:

- Single Reader Single Writer (SRSW)
- Multiple Reader Single Writer (MRSW)
- Multiple Reader Multiple Writer (MRMW)

We could classify registers by capacity:

- Boolean register (*true*/*false*), Integer register $(0, 1, ..., 2^N)$

We could classify registers by supported operations:

- Read-only, Write-once, Monotonically increasing, Arbitrary update
- Atomic read-modify-write operations

Mutual exclusion using read/write locations

- Correct, starvation-free algorithm exists (filter lock)
- Correct, starvation-free and fair algorithm exists (bakery algorithm)
- Any correct algorithm will require at least N registers with read and write operations

We could classify registers by allowed concurrency:

- Single Reader Single Writer (SRSW)
- Multiple Reader Single Writer (MRSW)
- Multiple Reader Multiple Writer (MRMW)

We could classify registers by capacity:

- Boolean register (*true*/*false*), Integer register $(0, 1, ..., 2^N)$

We could classify registers by supported operations:

- Read-only, Write-once, Monotonically increasing, Arbitrary update
- Atomic read-modify-write operations

Before diving into classifications (Lecture 7, Lecture 8), let's study few more mathematical instruments.

Homework: before final exam

When you finish all lectures of second block, you will know many advanced concepts:

- Concurrent object consistency definitions
- Register constructions
- Atomic snapshots
- Consensus number

Revise all the correctness proofs from this lecture and be ready to answer some tricky questions:

- In order to prove correctness of Peterson's algorithm, should we use MRSW or MRMW register?
- In order to prove correctness of mutual exclusion primitives, should we require all registers to be linearizable or weaker guarantees will fit?
- Could you adapt `FilterLock` to allow dynamic thread creation and deletion?

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Motivation

We need to formalize

- Safety (correctness)
- Liveness (progress)

Motivation

We need to formalize

- Safety (correctness)
- Liveness (progress)

for concurrent data structures and concurrent algorithms.

Motivation

We need to formalize

- Safety (correctness)
- Liveness (progress)

for concurrent data structures and concurrent algorithms.

- How to define safety and liveness "universally"?
- How to reason about whole programs / software components in isolation?
- How to compose modules and correctness proofs?
- Are there "better" or "simpler" approaches?

Concurrent objects

Goals:

- **Describe** object
- **Implement** object
- **Check** for errors

Concurrent objects

Goals:

- **Describe** object
- **Implement** object
- **Check** for errors

Check that **implementation** conforms to **specification**.

Concurrent objects

Goals:

- **Describe** object
- **Implement** object
- **Check** for errors

Check that **implementation** conforms to **specification**.
Let's focus on descriptions:

Concurrent objects

Goals:

- **Describe** object
- **Implement** object
- **Check** for errors

Check that **implementation** conforms to **specification**.
Let's focus on descriptions:

- when implementation is **correct**
- the conditions under which it guarantees **progress**

Sequential objects and specifications

Sequential object:

- State
  - Set of fields
- Methods
  - The only way to change state

Sequential objects and specifications

Sequential object:

- State
  - Set of fields
- Methods
  - The only way to change state

Sequential specification:

- If (**precondition**)
  - the object is in such-and-such a state
  - before you call the method
- Then (**postcondition**)
  - the method will return a particular value
  - or throw a particular exception
- and (**postcondition**)
  - the object will be in some other state
  - when the method returns

Sequential specifications

- Interactions among methods captured by side-effects on object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls
  - in sequential specification we need to only define object state between method calls

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls
  - in sequential specification we need to only define object state between method calls
- must characterize all possible interactions with concurrent calls

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls
  - in sequential specification we need to only define object state between method calls
- must characterize all possible interactions with concurrent calls
  - in sequential specification we described methods in isolation

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls
  - in sequential specification we need to only define object state between method calls
- must characterize all possible interactions with concurrent calls
  - in sequential specification we described methods in isolation
- everything can potentially interact with everything else

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object might never be between method calls
  - in sequential specification we need to only define object state between method calls
- must characterize all possible interactions with concurrent calls
  - in sequential specification we described methods in isolation
- everything can potentially interact with everything else
  - in sequential specification we could add new methods without affecting older methods

Sequential vs concurrent

Key difference (concurrency):

- Method call is not an event
- Method call is an interval

Implications:

- method calls could overlap, object
  - in sequential specification we ne
- must characterize all possible inte
  - in sequential specification we de
- everything can potentially interact
  - in sequential specification we cc

ods

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Consistency

What does it **mean** for a *concurrent* object to be correct?

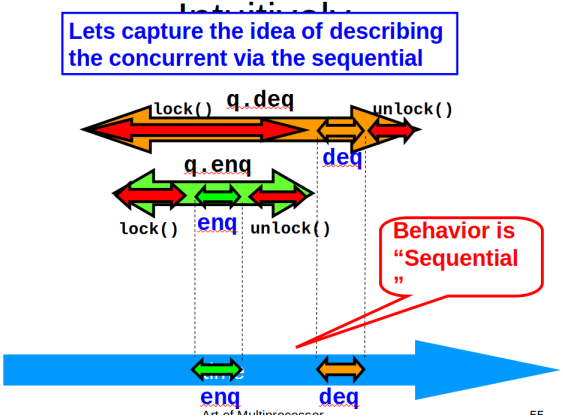
- What is a concurrent FIFO queue?

Consistency

What does it **mean** for a *concurrent* object to be correct?

- What is a concurrent FIFO queue?
- FIFO means strict temporal order
- Concurrent means ambiguous temporal order

Consistency



Linearizability

Each method should

- "take effect"
- instantaneously
- between invocation and response events

Linearizability

Each method should

- "take effect"
- instantaneously
- between invocation and response events

Object is correct if this "sequential" behavior is correct

Linearizability

Each method should

- "take effect"
- instantaneously
- between invocation and response events

Object is correct if this "sequential" behavior is correct
Any such concurrent object is **Linearizable**

Linearizability

Each method should

- "take effect"
- instantaneously
- between invocation and response events

Object is correct if this "sequential" behavior is correct
Any such concurrent object is **Linearizable**
Sounds like a property of an execution ...

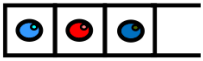
Linearizability

Each method should

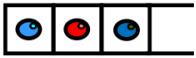
- "take effect"
- instantaneously
- between invocation and response events

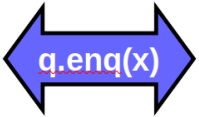
Object is correct if this "sequential" behavior is correct
Any such concurrent object is **Linearizable**
Sounds like a property of an execution ...
A **linearizable object**: one all of whose possible executions are linearizable

Examples

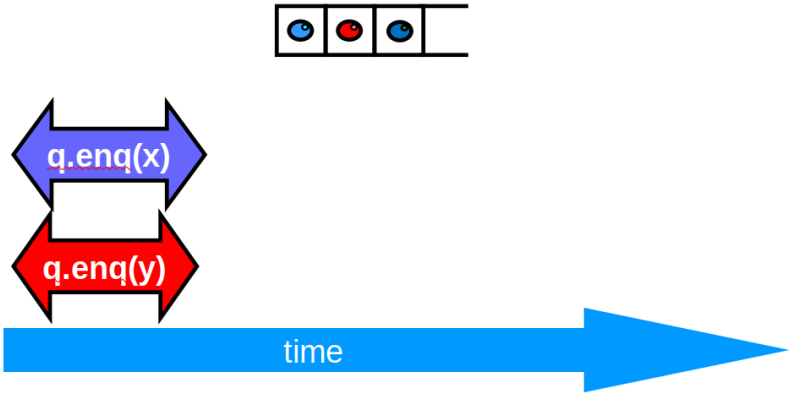


Examples

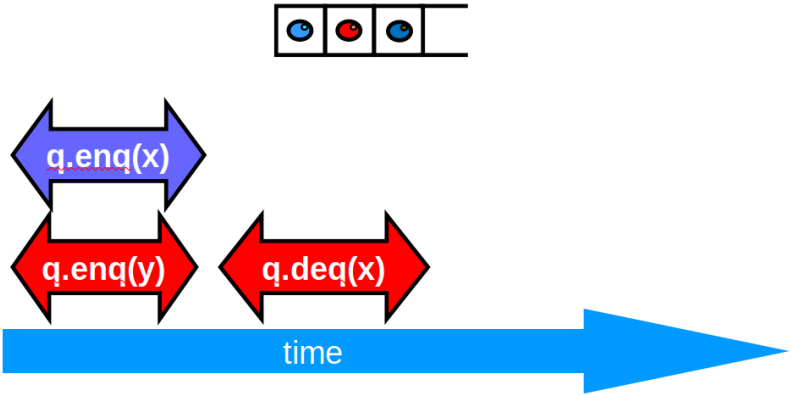




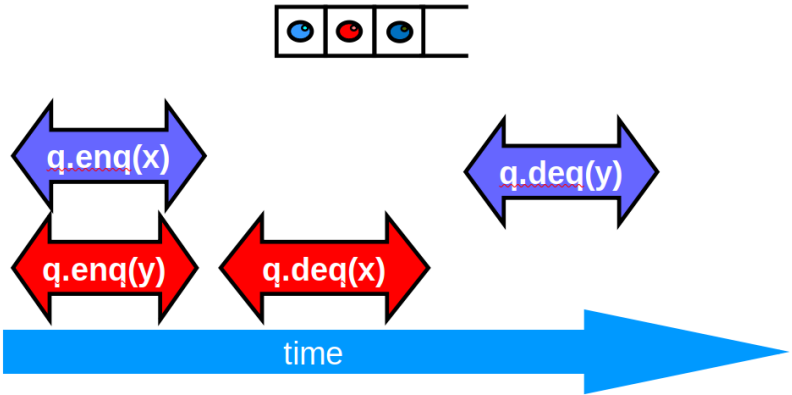
Examples



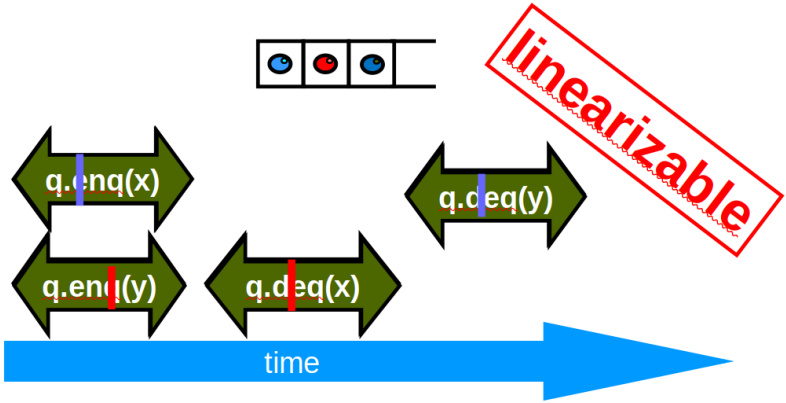
Examples



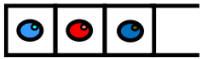
Examples



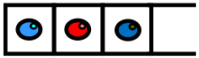
Examples

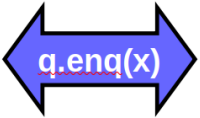


Examples

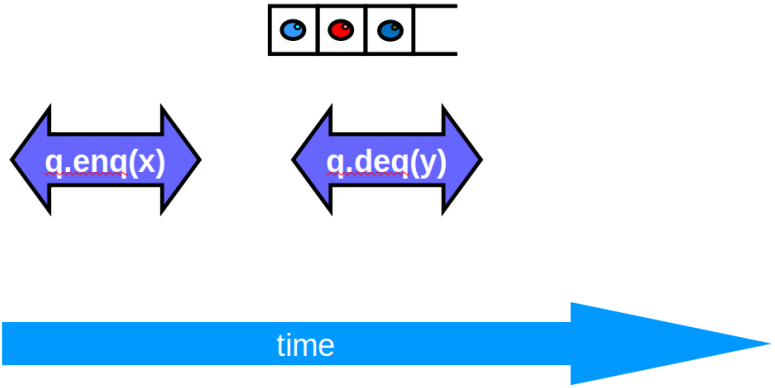


Examples

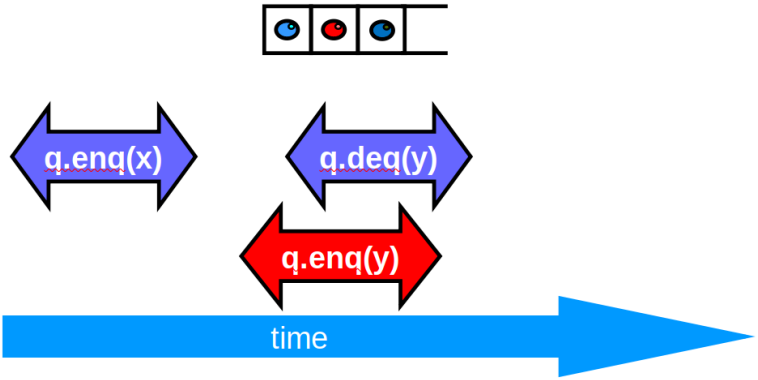




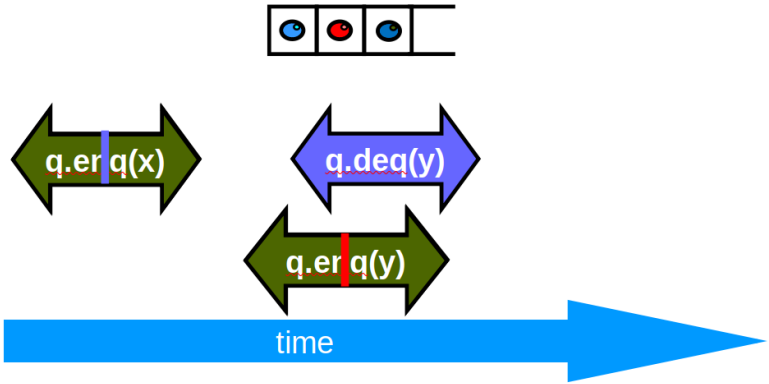
Examples



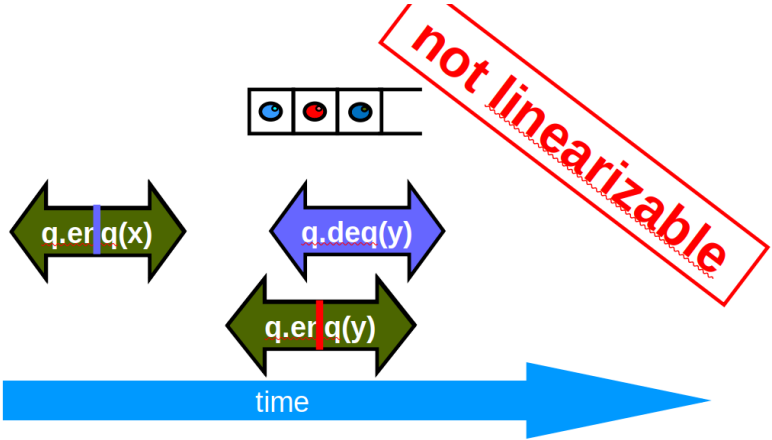
Examples



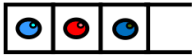
Examples



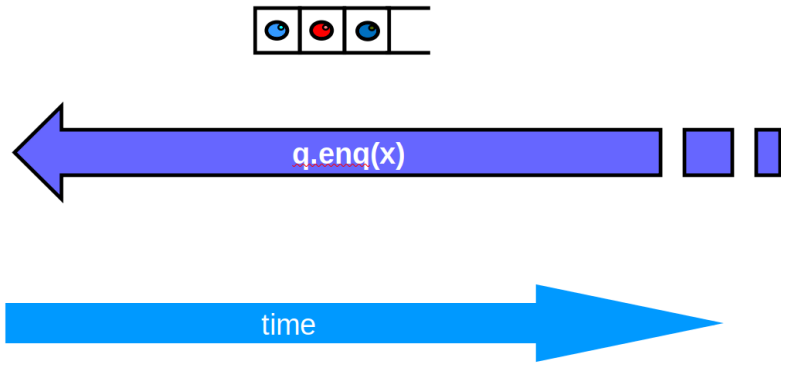
Examples



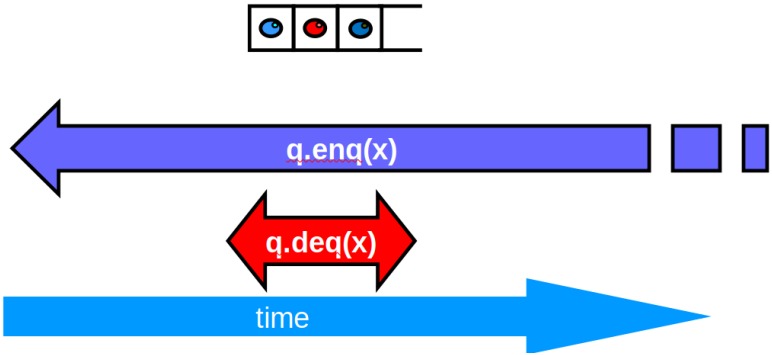
Examples



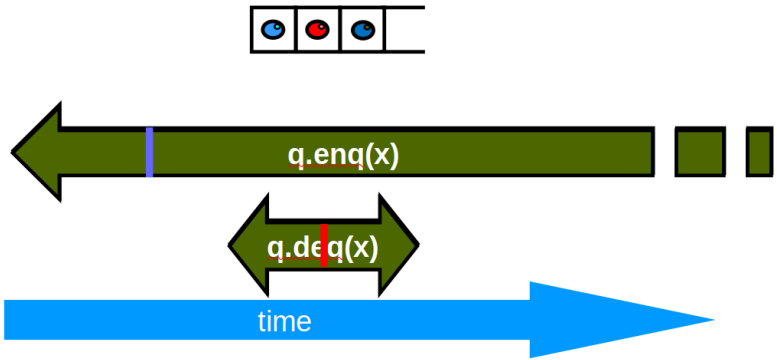
Examples



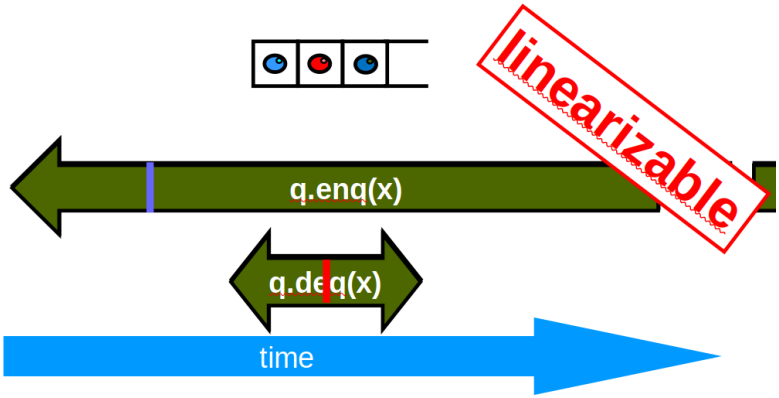
Examples



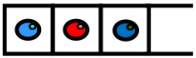
Examples

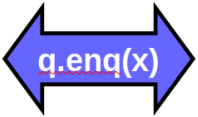


Examples

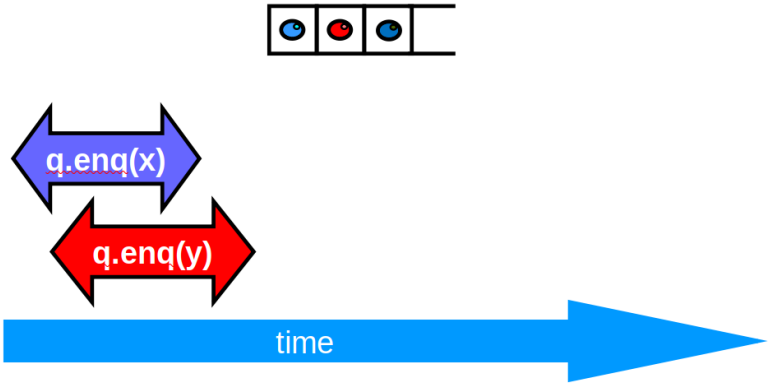


Examples

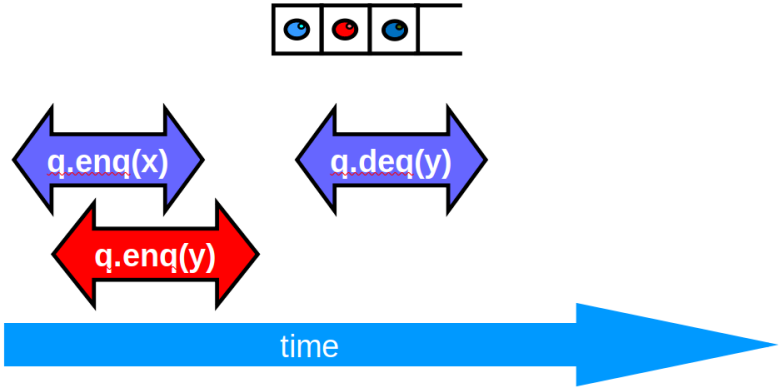




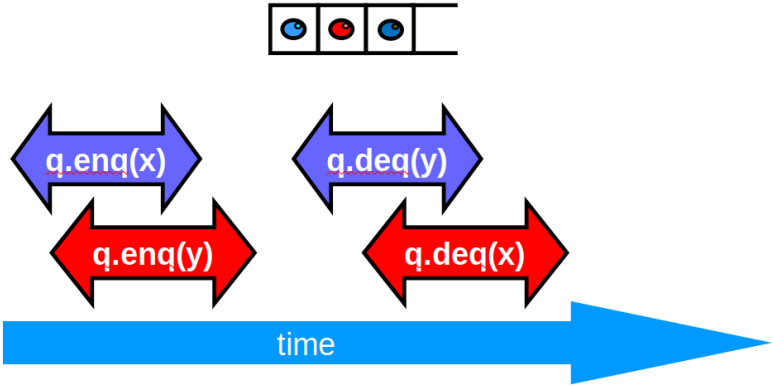
Examples



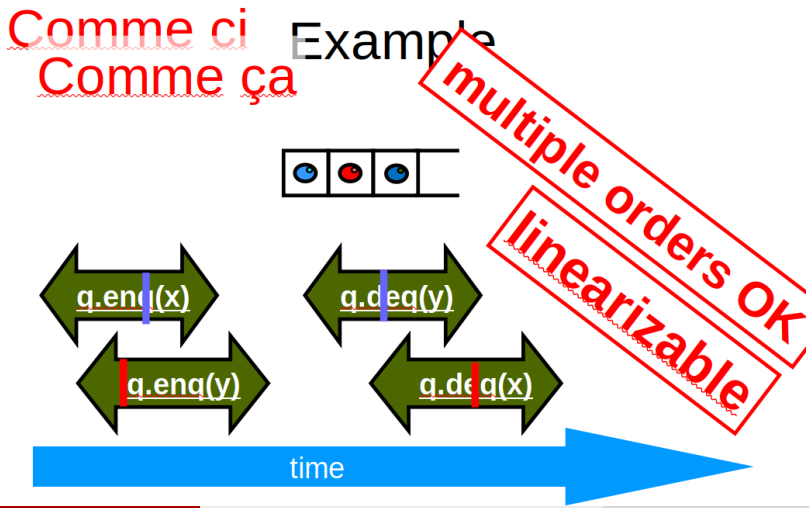
Examples



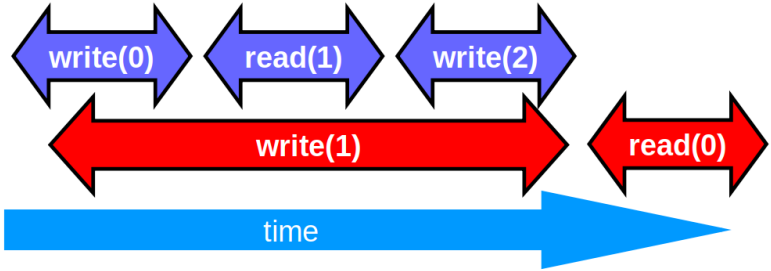
Examples



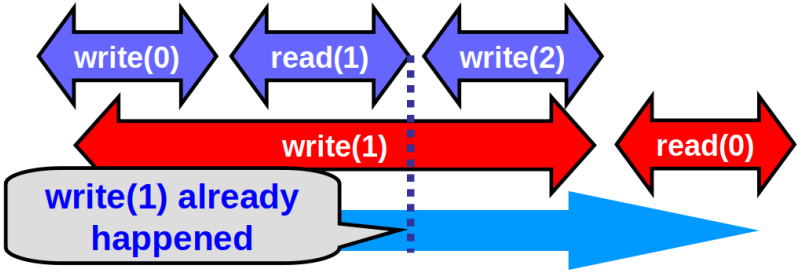
Examples


Comme ci
Comme ça
Example
multiple orders OK
linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time

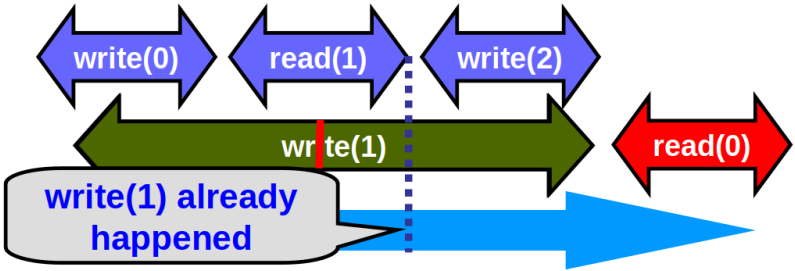
Examples



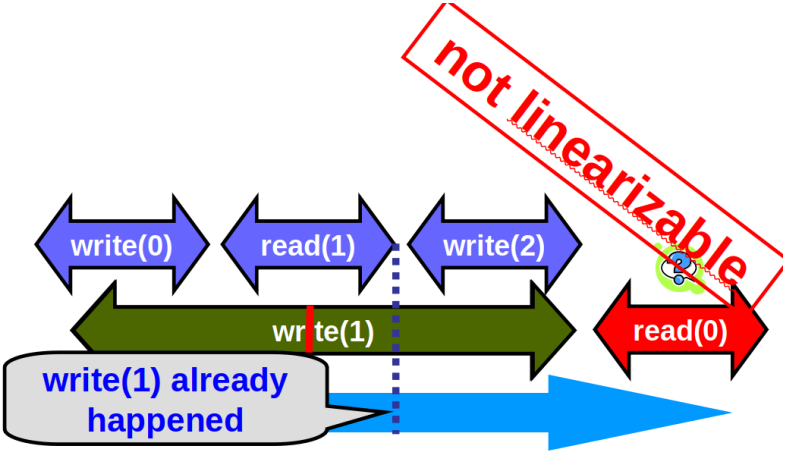
Examples



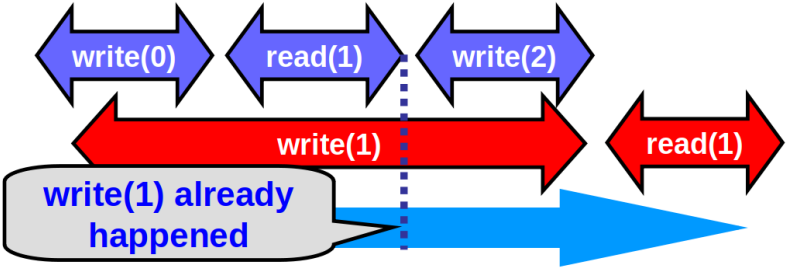
Examples



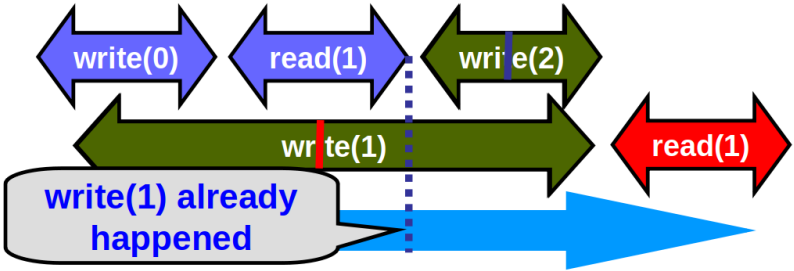
Examples


not linearizable
write(0)
read(1)
write(2)
write(1)
read(0)
write(1) already happened

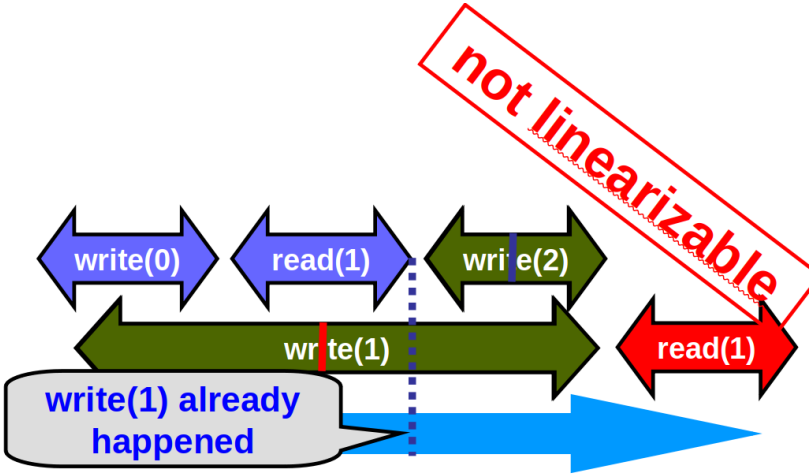
Examples



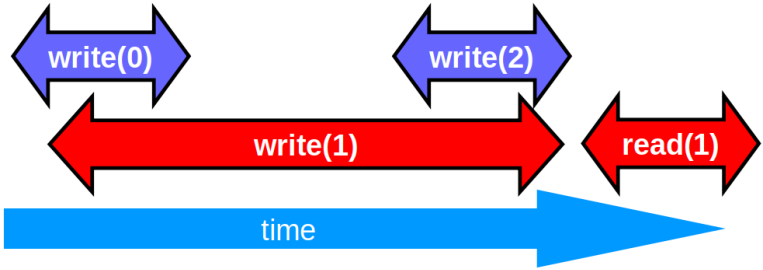
Examples



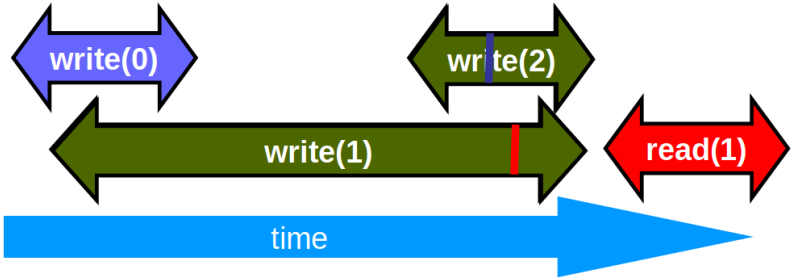
Examples


not linearizable
write(0)
read(1)
write(2)
write(1)
read(1)
write(1) already happened

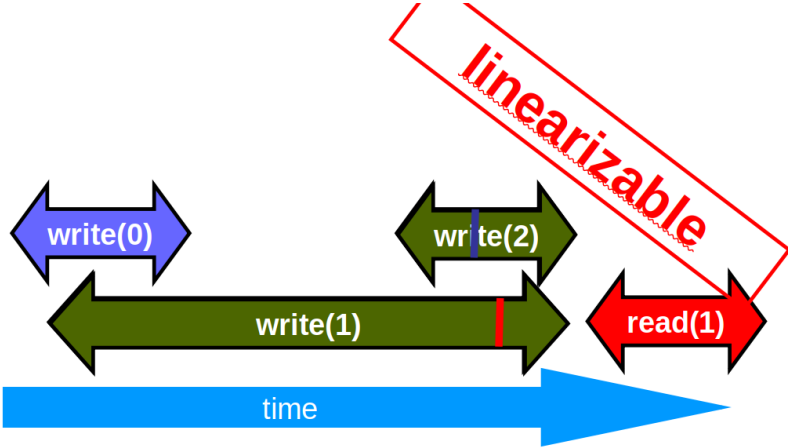
Examples



Examples



Examples



Linearizability: summary

Each method should

- "take effect"
- instantaneously
- between invocation and response events

Linearization point: when method "commit changes to object state"

- Could be different code points for different executions!

Linearizability

- requires to analyze executions, not just methods
- compositional – result of composing linearizable objects is linearizable[2]

[2] Optional homework: Theorem 3.6.1 in "The Art of Multiprocessor Programming"

Linearizability is not the only one

There exists:

Linearizability is not the only one

There exists:

- Sequential consistency
  - Provides other guarantees, harder to reason (no linearization points)
  - **Not** compositional
  - Hardware designers love to use weaker variations of it (Lecture 9 will be hard)

Linearizability is not the only one

There exists:

- Sequential consistency
  - Provides other guarantees, harder to reason (no linearization points)
  - **Not** compositional
  - Hardware designers love to use weaker variations of it (Lecture 9 will be hard)
- Quiescent consistency
  - One of the weakest (yet understandable) consistencies
  - Compositional
  - Complicated performance-oriented concurrent data structures may use it to formalize possible and impossible results

Linearizability is not the only one

There exists:

- Sequential consistency
  - Provides other guarantees, harder to reason (no linearization points)
  - **Not** compositional
  - Hardware designers love to use weaker variations of it (Lecture 9 will be hard)
- Quiescent consistency
  - One of the weakest (yet understandable) consistencies
  - Compositional
  - Complicated performance-oriented concurrent data structures may use it to formalize possible and impossible results

Linearizability is useful yet could be not applicable (by design[3] or by implementation[4])

[3] RCU: advantages and disadvantages https://en.wikipedia.org/wiki/Read-copy-update

[4] ConcurrentLinkedDeque is non-linearizable https://bugs.openjdk.org/browse/JDK-8256833

Intuition behind consistency definitions: takeaways

- **Describe** object first, then implement and test it
- Sequential specifications totally rock
  - Easy to understand and document
  - Modular and extensible
- Concurrent methods are not instantaneous and could overlap

Intuition behind consistency definitions: takeaways

- **Describe** object first, then implement and test it
- Sequential specifications totally rock
  - Easy to understand and document
  - Modular and extensible
- Concurrent methods are not instantaneous and could overlap

Solution 0: use locks and sequential specifications

Intuition behind consistency definitions: takeaways

- **Describe** object first, then implement and test it
- Sequential specifications totally rock
  - Easy to understand and document
  - Modular and extensible
- Concurrent methods are not instantaneous and could overlap

Solution 0: use locks and sequential specifications
Solution 1: mimic to sequential specification

- Points in code where method "commited changes" : linearization points
- Every concurrent execution have order among object-specific operations: linearizable object
- Provide specifications based on linearizability

Intuition behind consistency definitions: takeaways

- **Describe** object first, then implement and test it
- Sequential specifications totally rock
  - Easy to understand and document
  - Modular and extensible
- Concurrent methods are not instantaneous and could overlap

Solution 0: use locks and sequential specifications
Solution 1: mimic to sequential specification

- Points in code where method "commited changes" : linearization points
- Every concurrent execution have order among object-specific operations: linearizable object
- Provide specifications based on linearizability

Solution 2: use more fragile and complicated consistency definitions

Intuition behind consistency definitions: takeaways

- **Describe** object first, then implement and test it
- Sequential specifications totally rock
  - Easy to understand and document
  - Modular and extensible
- Concurrent methods are not instantaneous and could overlap

Solution 0: use locks and sequential specifications
Solution 1: mimic to sequential specification

- Points in code where method "commited changes" : linearization points
- Every concurrent execution have order among object-specific operations: linearizable object
- Provide specifications based on linearizability

Solution 2: use more fragile and complicated consistency definitions
Remember to check if your approach is compositional.

Lecture plan

1. Time, events, intervals, precedence
2. Mutual exclusion
   - 2 threads
   - N threads
   - Lower bounds on the Number of Locations
   - Optional: fairness
   - Mutual exclusion beyond read-write memory cells
3. Concurrent objects
   - Sequential and concurrent objects
   - Intuition behind consistency definitions
   - Optional: formal definitions and composability
4. Summary

Formal definitions

Optional homework: slides 95-176 from `chapter_03.ppt`
Optional homework: Chapter 3 "Concurrent Objects" up to section 3.6 "Formal Definitions" (pages 45 - 58)
Key insights:

- quiescent consistency
- sequential consistency
- linearizability
- formal definitions: history, projections
- compositional linearizability
- nonblocking property

Summary

First steps towards formalization of concurrent execution

- Timeline, Event, Interval, Precedence

Rigorously prove mutual exclusion properties:

- Mutual exclusion, Deadlock-freedom, Starvation-freedom

Non-trivial results for mutual exclusion

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

Formalizing allowed behaviour of concurrent object:

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

Summary: homework

Your second critical task will be related to the theoretical concurrency.
Maybe you would like to re-read slides and check out companion slides for "The Art of Multiprocessor Programming"