IN5170 Models of Concurrency

Lecture 3: Locks and Barriers

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

September 4, 2023

University of Oslo

Repetition

Repetition

- First general mechanisms and issues related to parallel programs
- *await language* and a simple version of the *producer/consumer* example
- Simple concurrency in Java

Repetition

- First general mechanisms and issues related to parallel programs
- *await language* and a simple version of the *producer/consumer* example
- Simple concurrency in Java

Today

Today

- Entry and exit protocols to *critical sections*
    - Protect reading and writing to shared variables
- *Barriers*
    - Iterative algorithms:
      Processes must *synchronize* between each iteration
    - Coordination using *flags*

Remember: Tightly coupled, synchronized Producer/Consumer

Await
```
int buf, p := 0; c := 0;
```

Await
```
process Producer {
 int a[N];   ...
 while(p < N){
   <await (p = c) >;
    buf := a[p];
    p := p+1;
 }
}
```

Await
```
process Consumer {
  int b[N];   ...
  while (c < N) {
    <await (p>c) >;
     b[c] := buf;
     c := c+1;
  }
}
```

- A strong invariant holds in *all states* in *all* executions of the program.
- *Global invariant*: c ≤ p ≤ c+1
- *Local invariant (Producer)*: 0 ≤ p ≤ N

Mutual Exclusion

Critical section

Critical Section

A *critical section* is part of the program that needs to be protected against interference by other processes

- Fundamental concept for concurrency - many solutions with different properties
- Execution under *mutual exclusion*
- Related to “atomicity”
- Using *locks* and low-level operations with software or hardware support

Critical section

Critical Section

A *critical section* is part of the program that needs to be protected against interference by other processes

- Fundamental concept for concurrency - many solutions with different properties
- Execution under *mutual exclusion*
- Related to "atomicity"
- Using *locks* and low-level operations with software or hardware support

How can we implement critical sections / conditional critical sections?

Access to Critical Section (CS)

- Basic scenario: Several processes compete for access to a shared resource
- Usage of the resource needs to be protected in a critical section
- Only one process can have access at a time (i.e., mutual exclusion)
  - Execution of bank transactions
  - Access to a printer or other resources
  - ...
- How to we control the *access* of processes to the CS?

Access to Critical Section (CS)

- Basic scenario: Several processes compete for access to a shared resource
- Usage of the resource needs to be protected in a critical section
- Only one process can have access at a time (i.e., mutual exclusion)
  - Execution of bank transactions
  - Access to a printer or other resources
  - ...
- How to we control the *access* of processes to the CS?

Await

If we can implement critical sections, then we can also implement **await** statements!

- Must have exclusive control over state to atomically evaluate guard
- Must be able to block other processes if guard holds

General patterns for critical sections

- inside the CS we have operations on shared variables.
- Access to the CS must then be protected to prevent interference.
- Coarse-grained pattern for *n* uniform processes repeatably executing some critical section

Await

```
process p[i=1 to n] {
  while (true) {
    CSentry          # entry protocol to CS
    CS
    CSexit           # exit protocol from CS
    non-CS
  }
}
```

- *Assumption*: A process which enters the CS will eventually leave it.

⇒ *Programming advice*: be aware of exceptions inside CS!

Naive solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {
  while (true) {
    while( in = 1 ) {skip };
    CS
    in := 2;
  }
}
```

Await

```
process p2 {
  while (true) {
    while( in = 2 ) {skip };
    CS
    in := 1;
  }
}
```

Naive solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {
  while (true) {
    while( in = 1 ) {skip };
    CS
    in := 2;
  }
}
```

Await

```
process p2 {
  while (true) {
    while( in = 2 ) {skip };
    CS
    in := 1;
  }
}
```

- *Entry protocol:* Busy waiting
- *Exit protocol:* Atomic assignment

Naive solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {
  while (true) {
    while( in = 1 ) {skip };
    CS
    in := 2;
  }
}
```

Await

```
process p2 {
  while (true) {
    while( in = 2 ) {skip };
    CS
    in := 1;
  }
}
```

- *Entry protocol:* Busy waiting
- *Exit protocol:* Atomic assignment

Discussion: what are the limitations of this solution?

Desired properties

1. **Mutual exclusion:** At any time, at most one process is inside CS.
2. **Absence of deadlock:** If all processes are trying to enter CS, at least one will succeed.
3. **Absence of unnecessary delay:** If some processes are trying to enter CS, while the other processes are in their non-critical sections, at least one will succeed.
4. **Eventual entry:** A process attempting to enter CS will eventually succeed.

Desired properties

1. **Mutual exclusion:** At any time, at most one process is inside CS.
2. **Absence of deadlock:** If all processes are trying to enter CS, at least one will succeed.
3. **Absence of unnecessary delay:** If some processes are trying to enter CS, while the other processes are in their non-critical sections, at least one will succeed.
4. **Eventual entry:** A process attempting to enter CS will eventually succeed.

Liveness and Safety in Critical Sections

The first three are *safety* properties. The last is a *liveness* property.

Reminder: Invariants and Atomic Sections

Global Invariants

A *global* invariant is a property over the shared variables that holds at every point during program execution (strong) or at every point outside an atomic section (weak)

- Safety property (something bad does not happen)
- Proof by induction: Holds initially and is preserved by every step

Atomic Sections

Statements grouped into a section that is always executed atomically.

- Conditional: <**await**(B) S>
- **await**(B) is known as condition synchronization, where B is evaluated atomically
- The whole block is executed atomically when B is true
- Unconditional: we write just < S >

Critical sections using "locks"

Await

```
bool lock := false;
process [i=1 to n] {
  while (true) {
    < await (!lock) lock := true >;
    CS;
    lock := false;
    non-CS
  }
}
```

Critical sections using “locks”

Await

```
bool lock := false;
process [i=1 to n] {
  while (true) {
    < await (!lock) lock := true >;
    CS;
    lock := false;
    non-CS
  }
}
```

Safety Properties

- Mutex
- Absence of deadlock and absence of unnecessary waiting

Can we remove the angle brackets $<$ ... $>$?

CS with AS: Test & Set (TAS)

Test & Set is a pattern for implementing a *conditional atomic action*:

Await

```
TS(lock) {
  < bool initial := lock;
  lock := true >;
  return initial
}
```

CS with AS: Test & Set (TAS)

Test & Set is a pattern for implementing a *conditional atomic action*:

Await

```
TS(lock) {
  < bool initial := lock;
  lock := true >;
  return initial
}
```

Effects of TS(lock)

- *Side effect:* The variable lock will always have value true after TS(lock),
- *Returned value:* true or false, depending on the original state of lock
- Exists as an atomic HW instruction on many machines.

Critical section with TS and spin-lock

Await

```
bool lock := false;

process p [i=1 to n] {
  while (true) {
    while (TS(lock)) {skip};        # entry protocol
    CS
    lock := false;                  # exit protocol
  }
}
```

Critical section with TS and spin-lock

Await

```
bool lock := false;

process p [i=1 to n] {
  while (true) {
    while (TS(lock)) {skip};       # entry protocol
    CS
    lock := false;                 # exit protocol
  }
}
```

- Safety: Mutex, absence of deadlock and of unnecessary delay.
- Strong fairness is needed to guarantee eventual entry for a process
- Problematic memory access pattern: lock as a hotspot

Reducing Writes

Test, Test and Set

Test, Test and Set (TTAS) reduces the number of writes by introducing more reads in the entry protocol.

Reducing Writes

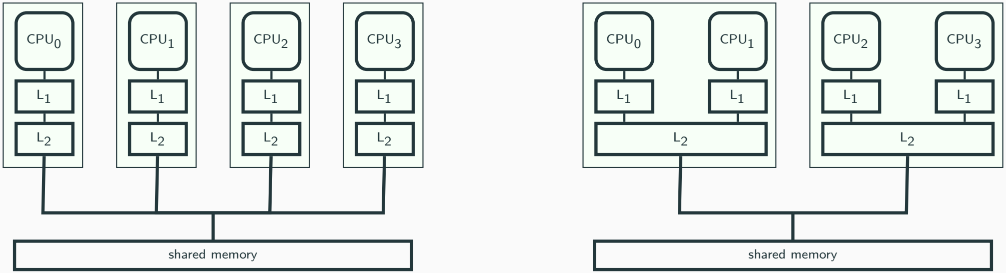
Test, Test and Set

Test, Test and Set (TTAS) reduces the number of writes by introducing more reads in the entry protocol.

Await

```
bool lock = false;
process p[i = 1 to n] {
  while (true) {
    while (lock) {skip};    # two additional spin lock checks
    while (TS(lock)) { while (lock) {skip} };
    CS;
    lock := false;
  }
}
```

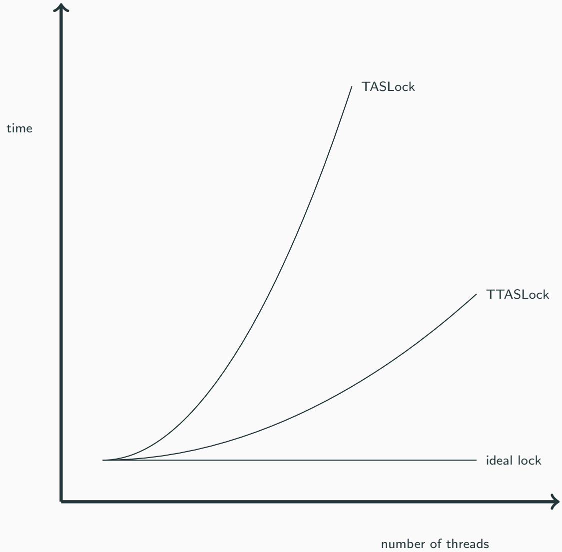
A glance at HW for shared memory



Contention

- TAS accesses main memory and synchronizes the caches through writing
- Reading can just access cache

Multiprocessor performance under load (contention)



Scheduling and Atomic Sections

Implementing await-statements

Let CSentry and CSexit implement entry- and exit-protocols to the critical section.

Unconditional Atomic Sections

The statement <S> can be implemented by

CSentry; S; CSexit;

Conditional Atomic Sections

Implementation of < **await** (B) S;> :

Await

```
CSentry;
while (!B) {CSexit ; CSentry};
S;
CSexit;
```

Implementation can be optimized with some delay between the exit and entry in the while body.

Scheduling and fairness - when processes shared a processor

- We want liveness properties as well, in particular *eventual entry*
- Eventual entry relies on scheduling and fairness

Enabledness

A statement is *enabled* in a state if the statement can in principle be executed next.

Await

```
bool x := true;
co while (x){ skip }; || x := false co
```

Scheduling

a strategy that for all points in an execution decides which enabled statement to execute.

Fairness (informally)

Enabled statements should not "systematically be neglected" by the scheduling strategy

Fairness notions

Possible status changes

- Disabled $\rightarrow$ enabled
- Enabled $\rightarrow$ disabled

In our language, only conditional atomic segments can have status changes

Different forms of fairness for different forms of statements

1. For statements that are always enabled
2. For those that once they become enable, they *stay enabled*
3. For those whose enabledness shows "on-off" behavior

Unconditional fairness

Definition (Unconditional fairness)

A scheduling strategy is *unconditionally fair* if each enabled unconditional atomic action, will eventually be chosen.

Await

```
bool x := true;
co  while (x){ skip }; || x := false co
```

- x := false is unconditional

⇒ The action will eventually be chosen

- Guarantees termination (in this example)
- A round robin scheduling strategy execution is unconditionally fair
- *Note:* loops and branchings are *not* conditional atomic statements

Weak fairness

Definition (Weak fairness)

A scheduling strategy is *weakly fair* if

- unconditionally fair
- every conditional atomic action will eventually be chosen, assuming that the condition becomes true and *remains true* until the action is executed.

Await

```
bool x = true, int y = 0;
co while (x) y := y + 1; || < await y ≥ 10; > x := false; oc
```

- When y $\geq$ 10 becomes true, this condition remains true
- This ensures termination of the program
- Here: again round robin scheduling

Strong fairness

Definition (Strongly fair scheduling strategy)

- unconditionally fair and
- each conditional atomic action will eventually be chosen, if the condition is true infinitely often.

Await

```
bool x := true; y := false;
co
     while (x) {y:=true; y:=false}
||
    < await(y) x:=false >
oc
```

- under strong fairness: y true ∞-often $\Rightarrow$ termination
- under *weak fairness:* non-termination possible

Fairness for critical sections using locks

The CS solutions shown need strong fairness to guarantee liveness, i.e., access for a given process (i):

Fairness for critical sections using locks

The CS solutions shown need strong fairness to guarantee liveness, i.e., access for a given process (i):

- Steady inflow of processes which want the lock
- value of `lock` alternates
  (infinitely often) between true and false

Challenges

- How to design a scheduling strategy that is both practical and strongly fair?
- Next part: How to design critical sections where eventual access is guaranteed for weakly fair strategies?

Weakly fair solutions for critical sections

- *Tie-Breaker Algorithm*
- *Ticket Algorithm*
- Others described in the literature

Tie-Breaker algorithm

Idea

- Requires no special machine instruction (like TS)
- We will look at the solution for two processes
- Each process has a private lock
- Each process sets its lock in the entry protocol
- The private lock is read, but is not changed by the other process

Naive solution

Await

```
int in = 1 # always 1 or 2
```

Await

```
process p1 {
  while (true) {
    while( in = 1 ) {skip };
    CS
    in := 2;
  }
}
```

Await

```
process p2 {
  while (true) {
    while( in = 2 ) {skip };
    CS
    in := 1;
  }
}
```

Naive Solutions: Problems

- Strict alternation
- No eventual entry for a single process
- Entry protocol: busy waiting
- Exit protocol: atomic assignment
- What about more than two processes?
- What about different execution times?

Tie-Breaker algorithm: Attempt 1

Await

```
Boolean in1 = false , in2 = false ;
```

Await

```
process p1 {
  while(true){
    while(in2) {skip};
    in1 := true;
    CS
    in1 := false;
  }
}
```

Await

```
process p2 {
  while(true){
    while(in1) {skip};
    in2 := true;
    CS
    in2 := false;
  }
}
```

Mutex not established, because both processes may be able to pass the entry protocol

What do we want as a global invariant?

Tie-Breaker algorithm: Attempt 2 (reordering)

Await

```
Boolean in1 = false, in2 = false;
```

Await

```
process p1 {
  while(true){
    in1 := true;
    while(in2) {skip};
    CS
    in1 := false;
  }
}
```

Await

```
process p2 {
  while(true){
    in2 := true;
    while(in1) {skip};
    CS
    in2 := false;
  }
}
```

Problem

Can deadlock if both variables are written before read.

Tie-Breaker algorithm: Attempt 3 (with await)

- Avoid deadlock through *tie-break* and decide for one process
- For fairness: do not always give priority to same specific process
- Add new variable: last to know which process last started the entry protocol

Await

```
Boolean in1 = false, in2 = false; Int last = 1;
```

Await

```
process p1 {
 while(true){
  in1 := true; last := 1;
  < await (!in2 || last = 2) >
  CS
  in1 := false;
 }}
```

Await

```
process p2 {
 while(true){
  in2 := true; last := 2;
  < await (!in1 || last = 1) >
  CS
  in2 := false;
 }}
```

Tie-Breaker algorithm

Even if the variables in1, in2 and last can change the value
while a wait-condition evaluates to true, the wait condition will *remain true*.

Tie-Breaker algorithm

Even if the variables in1, in2 and last can change the value
while a wait-condition evaluates to true, the wait condition will *remain true*.

p1 sees that the wait-condition is true:

- in2 $=$ false
    - in2 can eventually become true,
      but then p2 must also set last to 2
    - Then the wait-condition to p1 still holds

Tie-Breaker algorithm

Even if the variables in1, in2 and last can change the value
while a wait-condition evaluates to true, the wait condition will *remain true*.

p1 sees that the wait-condition is true:

- in2 $=$ false
    - in2 can eventually become true,
      but then p2 must also set last to 2
    - Then the wait-condition to p1 still holds
- last $= 2$
    - Then last $= 2$ will hold until p1 has executed

Tie-Breaker algorithm

Even if the variables in1, in2 and last can change the value
while a wait-condition evaluates to true, the wait condition will *remain true*.

p1 sees that the wait-condition is true:

- in2 $=$ false
  - in2 can eventually become true,
    but then p2 must also set last to 2
  - Then the wait-condition to p1 still holds
- last $= 2$
  - Then last $= 2$ will hold until p1 has executed

Thus we can replace the **await**-statement with a **while**-loop.

Tie-Breaker algorithm (4)

Await

```
Boolean in1 = false, in2 = false; Int last = 1;
```

Await

```
process p1 {
 while(true){
  in1 := true;
  last := 1;
  while (in2 && last != 2)
    skip;
  CS
  in1 := false;
 }}
```

Await

```
process p2 {
 while(true){
  in2 := true;
  last := 2;
  while (in1 && last != 1)
    skip;
  CS
  in2 := false;
 }}
```

Ticket algorithm

Multi-Tie-Breaker

- Generalizable to many processes (see book)
- But does not scale: If the Tie-Breaker algorithm is scaled up to n processes, we get a loop with $n - 1$ 2-process Tie-Breaker algorithms.

The *ticket algorithm* provides a simpler solution for critical sections for n processes.

- Intuition: ticket queue at old-fashioned government agencies
- A customer/process which comes in takes a number which is higher than the number of all others who are waiting
- The customer is served when a ticket window is available and the customer has the lowest ticket number.

Ticket algorithm: Sketch (n processes)

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
  while (true) {
    < turn[i] := number; number := number +1 >;
    < await (turn[i] = next)>;
    CS
    <next := next + 1>;
  }
}
```

- **await**-statement: can be implemented as while-loop
- turn[i] can be a local variable of process[i]
- Some machines have an *instruction* fetch-and-add (FA):
  FA(var, incr)= < **int** tmp := var; var := var + incr; return tmp;>

Ticket algorithm: Implementation

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
  while (true) {
    turn[i] := FA(number, 1);
    while (turn [i] != next) {skip};
    CS
    next := next + 1;
  }
}
```

- Without FA, we use an extra CS:

  CSentry; turn[i]:=number; number:= number + 1; CSexit;

- What is a *global* invariant for the ticket algorithm?

Ticket algorithm: Implementation

Await

```
int number := 1; next := 1; turn[1:n] := ([n] 0);
process [i = 1 to n] {
  while (true) {
    turn[i] := FA(number, 1);
    while (turn [i] != next) {skip};
    CS
    next := next + 1;
  }
}
```

- Without FA, we use an extra CS:

  CSentry; turn[i]:=number; number:= number + 1; CSexit;

- What is a *global* invariant for the ticket algorithm?

$$0 < next \leq number$$

Locks and Barriers

Barrier synchronization

- Computation of disjoint parts in parallel (e.g. array elements).
- Processes go into a loop where each iteration is dependent on the results of the previous.

Await

```
process Worker[i=1 to n] {
    while (true) {
      # perform task i;
      # barrier:
    }
}
```

Barrier synchronization

- Computation of disjoint parts in parallel (e.g. array elements).
- Processes go into a loop where each iteration is dependent on the results of the previous.

Await

```
process Worker[i=1 to n] {
    while (true) {
      # perform task i;
      # barrier:
    }
}
```

All processes must reach the barrier before any can continue.

Shared counter

A number of processes can synchronize the end of their tasks using a *shared counter*:

Await

```
int count := 0;
process Worker[i=1 to n] {
 while (true) {
  # perform task i
   < count := count+1>; < await(count = n)>;
  }
}
```

- Can be implemented using the FA instruction.

Disadvantages

- count must be reset between each iteration and is updated using atomic operations.
- Inefficient: Many processes read and write count concurrently.

Coordination using flags

- **Goal:** Avoid contention, i.e., too many read- and write-operations on one variable!

Coordination using flags

- **Goal:** Avoid contention, i.e., too many read- and write-operations on one variable!
- Divides shared counter into *several* variables, with one global coordinator process

Flag synchronization principles:

1. The process waiting for a flag is the one to reset that flag
2. A flag will not be set before it is reset

Coordination using flags

- **Goal:** Avoid contention, i.e., too many read- and write-operations on one variable!
- Divides shared counter into *several* variables, with one global coordinator process

Await

```
Worker[i]:
    # task i;
    arrive[i] := 1;
    < await (continue[i] = 1);>
Coordinator:
    for [i=1 to n] < await (arrive[i]=1);>
    for [i=1 to n] continue[i] := 1;
```

Flag synchronization principles:

1. The process waiting for a flag is the one to reset that flag
2. A flag will not be set before it is reset

Synchronization using flags

Both arrays continue and arrived are initialized to 0.

Synchronization using flags

Both arrays continue and arrived are initialized to 0.

Await

```
process Worker [i = 1 to n] {
 while (true) {
  # code to implement task i
  arrive[i] := 1;
  < await (continue[i] := 1>;
  continue[i]  := 0;
 }
}
```

Synchronization using flags

Both arrays continue and arrived are initialized to 0.

Await

```
process Worker [i = 1 to n] {
 while (true) {
  # code to implement task i
  arrive[i] := 1;
  < await (continue[i] := 1>;
  continue[i]  := 0;
 }
}
```

Await

```
process Coordinator {
 while (true) {
  for [i = 1 to n] {
    <await (arrived[i] = 1)>;
    arrived[i] := 0;
  };
  for [i = 1 to n] {
    continue[i] := 1;
  }
 }
}
```

Summary: Implementation of Critical Sections

Await

```
bool lock = false;
<await (!lock) lock := true >;   # entry protocol
# CS
<lock := false>                  # exit protocol
```

- Spin lock implementation of entry: `while (TS(lock)) skip`
- Exit without critical region.

Summary: Implementation of Critical Sections

Await

```
bool lock = false;
<await (!lock) lock := true >;  # entry protocol
# CS
<lock := false>                 # exit protocol
```

- Spin lock implementation of entry: `while (TS(lock)) skip`
- Exit without critical region.

Drawbacks:

- Busy waiting protocols are often complicated
- Inefficient if there are fewer processors than processes: wastes time executing an empty loop
- No clear distinction between variables used for synchronization and computation

Summary: Implementation of Critical Sections

Await

```
bool lock = false;
<await (!lock) lock := true >;  # entry protocol
# CS
<lock := false>                 # exit protocol
```

- Spin lock implementation of entry: `while (TS(lock)) skip`
- Exit without critical region.

Drawbacks:

- Busy waiting protocols are often complicated
- Inefficient if there are fewer processors than processes: wastes time executing an empty loop
- No clear distinction between variables used for synchronization and computation

Desirable to have special tools for synchronization protocols: semaphores

Locks in Java: Introduction

- How to ensure mutual exclusion in Java?
- The `java.util.concurrent.locks` package contains interfaces and classes for locking and waiting for conditions (distinct from built-in synchronisation/monitors)
- Manual lock management: flexible, but must be used cautiously

Lock interface

- Supports different semantics of locking
- Main implementation: *ReentrantLock*

ReadWriteLock Interface

- Locks that may be shared among readers but are exclusive to writers
- Only implementation: ReentrantReadWriteLock

Condition Interface

Condition (variables) associated with locks

Locks in Java: The Lock Interface (I)

Flexibility of locks $\Longrightarrow$ responsibility to use locks correctly

- Ensure that the lock is acquired before executing the code in the critical section
- Ensure that the lock is released in the end, even if something went wrong

Generic pattern for a method using a lock

Java

```java
Lock mutex = new Lock(); // shared between processes
...
mutex.lock();
try {
        ...             // critical section
} finally { mutex.unlock(); }
```

Locks in Java: The Lock Interface (II)

Includes methods:

- lock(): For acquiring the lock
- unlock(): For releasing the lock
- newCondition(): Returns a new Condition instance that it bound to this Lock instance

Example: A Counter

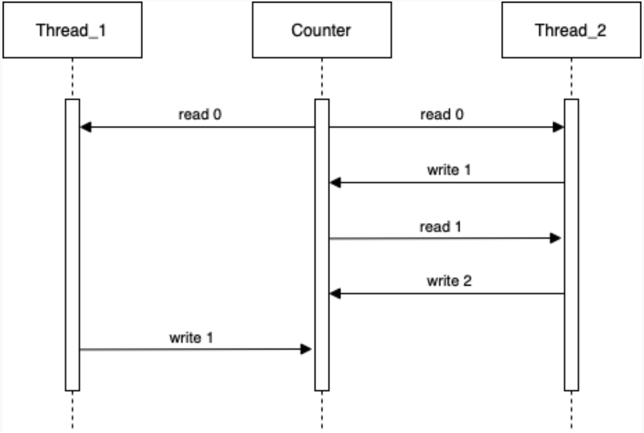
- Task: Add numbers from 0 to x using several threads
- Idea: Have a shared variable *value* that the threads increase

Java

```java
public class Counter {
        private int value;
        public Counter(int c) { value = c; }
        public int getAndIncrement() {
                int temp = value;
                value = temp + 1;
                return temp;
        }
}
```

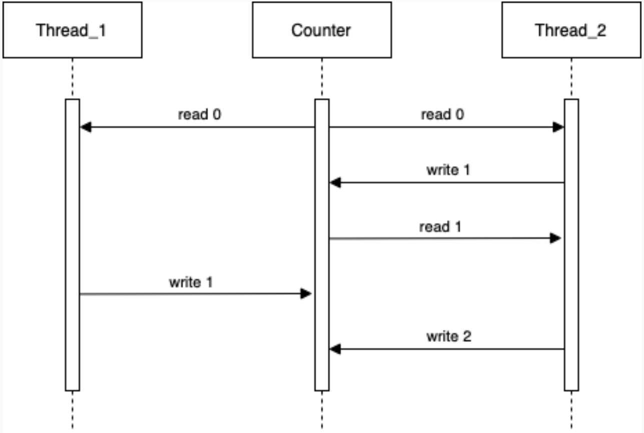
Example: A Counter (cont.)

Which values can *value* take if two threads call `getAndIncrement` three times in total?



Example: A Counter (cont.)

Which values can *value* take if two threads call `getAndIncrement` three times in total?



Java

```java
public class CounterLock {
  private int value;
  private Lock mutex = new ReentrantLock();
  public Counter(int c) { value = c; }
  public int getAndIncrement() {
    mutex.lock();              // entry
    int temp = 0;
    try {
      temp = value;            // critical section
      value = temp + 1;        // critical section
    } finally { mutex.unlock(); } //exit
    return temp;
  }
}
```

Repetition: Barrier Synchronization

Await

```
process Worker[i=1 to n] {
  while (true) {
    task i;
    wait until all n tasks are done      # barrier
  }
}
```

Barrier Synchronization

Barrier Synchronization in Java: The CyclicBarrier Class

- Waits for `n` arrivals before it unblocks
- After all threads have reached the barrier point, a Runnable command can be executed (BEFORE the threads are released)
- Called *cyclic* because it can be re-used after threads have been released

Barrier Synchronization in Java: The CyclicBarrier Class

Demo based on website

`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html`