


## Exercises for Tutorial H (Proposed solutions)

### Memory Models and Rust

**Hint:** These exercises were created by the tutors for the tutorial. They are neither relevant nor irrelevant for the exam. The evaluation of the difficulty is based on the assessment of the tutors.

## 1 Consistencies

### Exercise $\tau$ H.1


What's *sequential consistency*? What's *weak consistency*? What's *happens-before-consistency*? What do we need them for? What's the difference between a *program* and an *execution*? What do we mean with *compatibility*? What are *fence*-operations? 

#### Proposed Solutions $\tau$ H.1

The concrete definitions of the different kinds of consistencies can be looked up in the slides of *Lecture H1* (Definitions 28 - 31). They define memory models and with that describe which kind of executions are valid for a given program, concerning non-deterministic reordering effects coming from interleaving or the compiler. With that they abstractly state to the programmer, which concrete effects her program might have when executed. An execution is a partially ordered sequence of instructions stemming from a given program. The fundamental assumption for valid executions are that initialization operations must be ordered before thread-specific operations, that the *natural program order* must not be violated and that every operation must match an executable statement given by the program. In addition to this, an execution is compatible if it satisfies the fundamental assumptions, and the restrictions given by the underlying memory model.

A *fence*-operation is part of programs utilizing the *Weak Consistency with Fences*. They add structure to the program and may be used to prevent suspicious and unintuitive but weak consistent executions as *Out-of-Thin-Air-Reads*. Given a read-operation, if its respective write-operation is ordered after the fence, then also the read itself must be ordered after the fence.

### Exercise $\tau$ H.2 (*Consistency*)

Consider the following program and observations. Are they explainable with 

- sequential consistency?
- weak consistency?

If yes: State the execution and ordering to proof it.

For all cases where the observation is weak but not sequentially explainable: Is it possible to insert a *fence*-instruction such that "unexpected" behavior is not explainable with weak consistency (without fences) anymore?

	x = 0, y = 0			x = 0, y = 0	
(a)	1 r1 = x;	1 r2 = y;	(b)	1 r1 = y;	1 r2 = x;
	2 y = r1;	2 x = r2;		2 x = r1;	2 r3 = x;
(i)	x = 0, y = 0				3 if (r2 == r3) {
(ii)	x = 1337, y = 0				4 y = 2;
(iii)	x = 1337, y = 1337				5 }
			(i)	r1 = 2, r2 = 2, r3 = 2	
			(ii)	r1 = 0, r2 = 0, r3 = 2	
			(iii)	r1 = 0, r2 = 0, r3 = 0	

	$x = 0, y = 0, z = 0$	
(c)	<pre> 1 r2 = y; 2 r1 = x; 3 4 if (r1 &lt; r2) { 5   z = 10; 6 } </pre>	<pre> 1 x = 5; 2 y = 5; </pre>
(i)	$z = 10$	

	$x = 0, y = 5$	
(d)	<pre> 1 r1 = x; 2 r2 = y; 3 x = r1 + r2; </pre>	<pre> 1 if (x &gt; 0) { 2   r3 = y; 3   y = r3 * 2; 4 } </pre>
(i)	$x = 5, y = 5$	(iii) $x = 5, y = 20$
(ii)	$x = 5, y = 10$	(iv) $x = 10, y = 10$

### Proposed Solutions $\tau H.2$

- (a) (i) sequential and weak consistent  
(ii) inconsistent  
(iii) only weak consistent

It is possible to insert a **fence**-instruction after  $r1 = x$ .

- (b) (i) only weak consistent

It is possible to insert a **fence**-instruction after  $r1 = y$ .

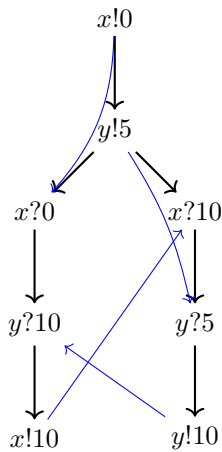
- (ii) inconsistent  
(iii) sequential and weak consistent

- (c) (i) only weak consistent

It is possible to insert a **fence**-instruction after  $x = 5$ .

- (d) (i) sequential and weak consistent  
(ii) sequential and weak consistent  
(iii) inconsistent  
(iv) only weak consistent

The following diagram explains the observation:



It is possible to insert a **fence**-instruction after  $r2 = y$ .

### Exercise $\tau H.3$ ( $\tau H.2$ -happens-before- $\tau H.3$ )

Consider the programmes from exercise  $\tau H.2$ , where you used **fence**-operations.

Can you achieve the same effect using *locks*?



### Proposed Solutions $\tau H.3$

You can get the same result by enclosing all lines of both agents on the left and on the right using a single lock.

### Exercise $\tau H.4$ (Agent 001 on a secret mission)

Consider the following pseuCo program:



<pre> 1 int x, y; 2 3 void a1() { 4     x = -1; 5     if (x &gt; 0) { 6         y = 1; 7     } 8     x = y; 9 }</pre>	<pre> 9 mainAgent { 10     start(a1()); 11 12     y = x; 13     if (y &gt; 0) { 14         x = 1; 15     } 16 }</pre>
-----------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------

- (a) Write out the program in the familiar table like format
- (b) Which of the following assignments of variables are possible after execution according to *sequential consistency*? For all possible assignments, give a valid execution and order that proves your claim.
 

(i)  $x = 0, y = 0$

(iii)  $x = 0, y = -1$

(v)  $x = 1, y = 0$

(vii)  $x = 1, y = 1$

(ii)  $x = -1, y = 0$

(iv)  $x = -1, y = -1$

(vi)  $x = 0, y = 1$
- (c) Which of the above assertions are additionally possible according to *weak consistency*? Again, give a valid execution and order for each assertion.
- (d) Are there any other possible assertions that can be proven according to *weak consistency*? If so, which ones?

### Proposed Solutions $\tau H.4$

$x = 0, y = 0$	
<pre> 1 x = -1; 2 r1 = x; 3 if (r1 &gt; 0) { 4     y = 1; 5 } 6 r2 = y; 7 x = r2;</pre>	<pre> 1 r3 = x; 2 y = r3; 3 r4 = y; 4 if (r4 &gt; 0) { 5     x = 1; 6 }</pre>

- (b) According to *sequential consistency*, only the following three assignments are possible:

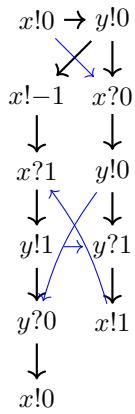
- $x = 0, y = 0$
- $x = -1, y = -1$
- $x = 0, y = -1$

You can view a valid execution and order using the function *Memory Model* and the following (semantically equivalent) pseuCo program: [pseuCo.com/#/edit/remote/8xkrtkiixmjf9fg15l85](https://pseuCo.com/#/edit/remote/8xkrtkiixmjf9fg15l85)

- (c) In addition to the above allocations, the following are also possible:

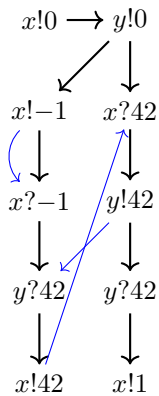
- $x = 0, y = 1$
- $x = 1, y = 0$
- $x = 1, y = 1$

Consider the following execution for this purpose:



- (d) Due to the lines 7 and 12 there is an *Out-Of-Thin-Air-Read*, whereby  $x$  and  $y$  can take arbitrary values.

Consider the following, exemplary execution:



### Exercise $\tau$ H.5 (001, 002 and 003 in the face of death)

$x = 0, y = 0, z = 0$



Take a look at the following program:

<pre> 1 r1 = x; 2 y = r1; 3 r2 = x; 4 if (r2 &lt; 0) { 5     z = r2 - 1; 6 } </pre>	<pre> 1 r3 = y; 2 z = r3; 3 r4 = y; 4 if (r4 &lt; 0) { 5     x = r4 - 1; 6 } </pre>	<pre> 1 r5 = z; 2 x = r5; 3 r6 = z; 4 if (r6 &lt; 0) { 5     y = r6 - 1; 6 } </pre>
-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------

- Which assignments of the variables  $x$ ,  $y$  and  $z$  are possible under *sequential consistency*?
- How does your answer change if we replace every  $<$  with a  $\leq$ ?
- Which assignment are possible under *weak consistency*? Can you save the program by using **fences**? How many do you need? How many **lock-unlock**-pairs do you need to enforce the same possibilities as in *sequential consistency*?

### Proposed Solutions $\tau$ H.5

- only  $x = 0, y = 0, z = 0$
- now  $x = 0, y = -1, z = -2$  and every permutation of it are possible
- There is a threeway circular dependency here which can, among other things, lead to *out-of-thin-air-reads*. Using a single **fence** each between line 1 and 2 in an arbitrary agent lets us fix the issue. One **lock-unlock**-pair each in two arbitrary agents wrapped around one reading and one writing access on the same global variable solves the problem.

## 2 Rust

### Exercise $\tau$ H.6 (Understanding ownership)

Remember that in Rust we have three kinds of types:

```
1 let mut v = vec![1, 2, 3]; // Owned: Vec<i32>
2 let v_ref = &v;           // Shared reference: &Vec<i32>
3 let v_mut_ref = &mut v;    // Mutable / exclusive reference: &mut Vec<i32>
```

Do the following programs compile? Why (not)?

(a)

```
1 fn dummy(_v: Vec<i32>) { /* nop */ }
2 fn main() {
3     let v = vec![1, 2, 3];
4     dummy(v);
5     v[0] = 42;
6 }
```

(d)

```
1 fn main() {
2     let mut v = vec![1, 2, 3];
3     for i in &v {
4         *i += 2;
5     }
6 }
```

(b)

```
1 fn dummy(_v: &Vec<i32>) { /* nop */ }
2 fn main() {
3     let v = vec![1, 2, 3];
4     dummy(&v);
5     v[0] = 42;
6 }
```

(e)

```
1 fn main() {
2     let mut v = vec![1, 2, 3];
3     for i in &mut v {
4         *i += 2;
5     }
6 }
```

(c)

```
1 fn dummy(_v: &mut Vec<i32>) { /* nop */ }
2 fn main() {
3     let mut v = vec![1, 2, 3];
4     dummy(&mut v);
5     v[1337] = 42;
6 }
```

(f)

```
1 fn main() {
2     let mut v = vec![1, 2, 3];
3     for i in &v {
4         v[0] = *i;
5     }
6 }
```

### Proposed Solutions $\tau$ H.6

- (a) No, `dummy()` takes ownership of `v`. Hence, in line 5, we cannot access `v` anymore.
- (b) No, the assignment `v[0] = 42` needs to borrow `v` mutably, but it is not declared as mutable. If we change `let` to `let mut` in line 3, everything is fine.
- (c) Yes. However, the program will panic at runtime as index 1337 is out of bounds.
- (d) No, `i` has type `&i32`, so we cannot modify it.
- (e) Yes.
- (f) No. In line 3, we create a shared reference to `v`, however in line 4, we need an exclusive reference. Of course, we cannot have both at the same time.

### Exercise $\tau$ H.7 (Counting references)

A key property of ownership and references in Rust is that it avoids both use-after-free bugs as well as memory leakage. The idea is that no reference may outlive the owned value itself. Furthermore, the owner is responsible for the cleanup. But what if there is no clear ownership relation in your program? What if some piece of data is collectively owned by the server threads and may only be freed after all servers have terminated? Is there some container in the Rust standard library that can help?

### Proposed Solutions $\tau$ H.7

Yes, in the concurrent setting we can use `std::sync::Arc`. ‘Arc’ stands for ‘Atomically Reference Counted’. (There is also `std::rc::Rc` for use in single-threaded settings.) The idea behind reference counting is just what the name says, counting the number of references to some piece of data. If you call `clone()` on such a reference, it does not clone the data itself but increments the reference counter by one. If a reference is dropped, the counter is decremented. If the counter reaches zero, the data can safely be deallocated (this is also a must to prevent memory leaks).

### Exercise $\tau$ H.8 (Back to the roots)

What does the following Rust program do? Translate it to `pseuCo!`



```
1 use std::sync::mpsc::{channel, Receiver, Sender};
2 use std::thread::spawn;
3
4 fn printer(rx: Receiver<u32>) {
5     while let Ok(v) = rx.recv() {
6         if v == u32::MAX {
7             return;
8         }
9         println!("{v}");
10    }
11 }
12
13 fn sieve(n: u32, prev_rx: Receiver<u32>, print_tx: Sender<u32>) {
14     let (next_tx, next_rx) = channel();
15
16     while let Ok(v) = prev_rx.recv() {
17         if v == u32::MAX {
18             match print_tx.send(u32::MAX) {
19                 Ok(_) => {
20                     //do nothing
21                 }
22                 Err(e) => {
23                     panic!("Not possible to send! Error: {}", e);
24                 }
25             }
26             return;
27         }
28         if v % n != 0 {
29             print_tx.send(v).unwrap();
30             spawn(move || sieve(v, next_rx, print_tx));
31             break;
32         }
33     }
34
35     while let Ok(v) = prev_rx.recv() {
36         if v == u32::MAX {
37             next_tx.send(u32::MAX).unwrap();
38             return;
39         }
40         if v % n != 0 {
41             next_tx.send(v).unwrap();
42         }
43     }
44 }
45
46 fn main() {
47     let (tx0, rx0) = channel();
48     let (print_tx, print_rx) = channel();
49     spawn(move || sieve(2, rx0, print_tx));
50     let p = spawn(move || printer(print_rx));
51
52     for i in 2..1000 {
53         tx0.send(i).unwrap();
54     }
55     tx0.send(u32::MAX).unwrap();
56
57     p.join().unwrap();
58     println!("finished!");
59 }
```

### Proposed Solutions $\tau H.8$


It prints the prime numbers  $< 1000$ . For the `pseuCo` program, see [pseuCo.com/#/edit/remote/zwirzvzgwkl4iyca0xok](https://pseuCo.com/#/edit/remote/zwirzvzgwkl4iyca0xok). Just a few remarks on the Rust code:

- We use the channels from the standard library. These are located in `std::sync::mpsc`. `mpsc` stands for “multiple processors, single consumer”. Unlike our `pseuCo` channels, we do not have a destructive multicast but a unicast. While we could call `tx.clone()` for some sending part `tx`, this is not possible for a receiving part.
- You might wonder about all these `unwrap()` calls. Rust does not have exceptions. Instead, all error handling is performed via return values, usually of type `Result<T, E>`. This is an enumeration type with two constructors: `Ok(T)` and `Err(E)` (like in SML, constructors can have arguments). Operations like sending or receiving from a channel might fail (that is, if the other side is disconnected). In this case, an error is returned. However, in our program, we are pretty sure that this will never happen. And if it did, we’d rather want the entire program to panic. This is, what `unwrap()` does.

Another way of dealing with results is to perform pattern matching on them, like we do at the while loops. We only continue with the loop body, if the value is of shape `Ok(v)`. If it is, `v` is bound to the inner value. As you might imagine, there also is an `if let`.

- Another thing that might seem strange is the exclamation mark in `println!()`. This just denotes that `println!()` is a macro. In a strictly type-safe programming language, variadic arguments we know from C’s `printf()` are not possible, unless you have runtime type information. However, carrying type information around at runtime is not for free. That is why the designers of Rust decided to avoid it. Instead, they designed a powerful preprocessor to translate the macros like `println!()` into code that does not need variadic arguments.
- If you are interested in Rust and want to learn more about it, have a look at <https://doc.rust-lang.org/book/>.
- You can use the *Rust Playground* to execute the program without installing rust: <https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=f4a6944f1c53b60623f3080ee21bb9c3>

### Exercise $\tau H.9$ (How to Err in crab language)

In Java we throw errors and also need to catch them. This is bad (as a rustacean would say)! Rust encodes possible failure inside the type system. There are three possible ways of representing this: 

- `panic!()` which completely halts the program.
- `Result<T, E>` which is an Enum with variants `Ok(T)` and `Err(E)` where `T` and `E` are Generics.
- `Option<T>` which is an Enum with variants `Some(T)` and `None` where `T` is a Generic.

For each of the following statements find one of the three variants to which it fits best.

- |                                       |                                                       |
|---------------------------------------|-------------------------------------------------------|
| (a) Continuing after is incorrect.    | (c) Success is expected and failure is the exception. |
| (b) Success and Failure are expected. | (d) Absolutely unexpected error.                      |

### Proposed Solutions $\tau H.9$

- |                                                                             |                                                                 |
|-----------------------------------------------------------------------------|-----------------------------------------------------------------|
| (a) <code>panic!()</code> as if you panic you should not recover but crash. | (c) <code>Result&lt;T, E&gt;</code> , see “I/O” in std crate.   |
| (b) <code>Option&lt;T&gt;</code> , see Vector access.                       | (d) <code>panic!()</code> as a panic is like an emergency stop. |

### Exercise $\tau$ H.10 (It's a match <3)

After you have learned about `Result<T, E>` and `Option<T>` we want to actually work with them. In rust this is done by the `match` keyword which kind of works like a supercharged switch case but with pattern matching. See here for details: [https://doc.rust-lang.org/rust-by-example/flow\\_control/match.html](https://doc.rust-lang.org/rust-by-example/flow_control/match.html)



- (a) Fill in the question mark so that the program prints the result or "Cannot divide by 0"

<pre>1 fn divide(numerator: f64, denominator: f64) -&gt;   ↳ Option&lt;f64&gt; { 2     if denominator == 0.0 { 3         None 4     } else { 5         Some(numerator / denominator) 6     } 7 }</pre>	<pre>1 // The return value of the function   ↳ is an option 2 let result = divide(2.0, 3.0); 3 // Pattern match to retrieve the   ↳ value 4 match result { 5     /// 6 }</pre>
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

- (b) Fill in the question mark so that the program either prints the value ( $\neq 42$ ) or prints the error without defining a new string.

```
1 fn no_42_allowed(argument: usize) -> Result<usize, &'static str> {  
2     if argument == 42 {  
3         Err("This cannot be the answer!")  
4     } else {  
5         Ok(argument)  
6     }  
7 }  
8 let example: usize = 42;  
9 match no_42_allowed(example) {  
10    ///  
11 }
```

### Proposed Solutions $\tau$ H.10

- (a)

```
1 fn divide(numerator: f64, denominator: f64) -> Option<f64> {  
2     if denominator == 0.0 {  
3         None  
4     } else {  
5         Some(numerator / denominator)  
6     }  
7 }  
8  
9 let result = divide(2.0, 3.0);  
10  
11 match result {  
12     Some(x) => println!("Result: {x}"),  
13     None    => println!("Cannot divide by 0"),  
14 }
```

- (b)

```
1 fn no_42_allowed(argument: usize) -> Result<usize, &'static str> {  
2     if argument == 42 {  
3         Err("This cannot be the answer!")  
4     } else {  
5         Ok(argument)  
6     }  
7 }  
8  
9 let example: usize = 42;  
10  
11 match no_42_allowed(example) {  
12     Ok(x) => println!("{}", x),  
13     Err(x) => println!("{}", x)  
14 }
```