Exercises for Tutorial H
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 τH.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?

Exercise τH.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?

(a)

x = 0, y = 0

```
1 r1 = x;      | 1 r2 = y;
2 y = r1;      | 2 x = r2;
```

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

(ii) $x = 1337, \quad y = 0$

(iii) $x = 1337, \quad y = 1337$

(b)

x = 0, y = 0

```
1 r1 = y;      | 1 r2 = x;
2 x  = r1;     | 2 r3 = x;
               | 3 if (r2 == r3) {
               | 4   y = 2;
               | 5 }
```

(i) $r1 = 2, \quad r2 = 2, \quad r3 = 2$

(ii) $r1 = 0, \quad r2 = 0, \quad r3 = 2$

(iii) $r1 = 0, \quad r2 = 0, \quad r3 = 0$

(c)

x = 0, y = 0, z = 0

```
1 r2 = y;      | 1 x = 5;
2 r1 = x;      | 2 y = 5;
3              |
4 if (r1 < r2) {
5   z = 10;
6 }
```

(i) $z = 10$

(d)

x = 0, y = 5

```
1 r1 = x;      | 1 if (x > 0) {
2 r2 = y;      | 2   r3 = y;
3 x  = r1 + r2;| 3   y  = r3 * 2;
               | 4 }
```

(i) $x = 5, \quad y = 5$

(ii) $x = 5, \quad y = 10$

(iii) $x = 5, \quad y = 20$

(iv) $x = 10, \quad y = 10$

Exercise τH.3 *(τH.2-happens-before-τH.3)*
Consider the programms from exercise τH.2, where you used `fence`-operations.

Can you achieve the same effect using *locks*?

Exercise τH.4 *(Agent 001 on a secret mission)*
Consider the following pseuCo program:

```
int x, y;

void a1() {
    x = -1;
    if (x > 0) {
      y = 1;
    }
    x = y;
}
```

```
mainAgent {
    start(a1());

    y = x;
    if (y > 0) {
      x = 1;
    }
}
```

(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) $x = 0, y = 0$ (ii) $x = -1, y = 0$ (iii) $x = 0, y = -1$ (iv) $x = -1, y = -1$ (v) $x = 1, y = 0$ (vi) $x = 0, y = 1$ (vii) $x = 1, y = 1$

(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?

Exercise τH.5 *(001, 002 and 003 in the face of death)*

Take a look at the following program:

`x = 0, y = 0, z = 0`

```
r1 = x;
y = r1;
r2 = x;
if (r2 < 0) {
   z = r2 - 1;
}
```

```
r3 = y;
z = r3;
r4 = y;
if (r4 < 0) {
   x = r4 - 1;
}
```

```
r5 = z;
x = r5;
r6 = z;
if (r6 < 0) {
   y = r6 - 1;
}
```

(a) Which assignments of the variables x, y and z are possible under *sequential consistency*?

(b) How does your answer change if we replace every $<$ with a $\leq$?

(c) Which assignment are possible under *weak consistency*? Can you save the program by using `fence`s? How many do you need? How many `lock`-`unlock`-pairs do you need to enforce the same possibilities as in *sequential consistency*?

2 Rust

Exercise τH.6 *(Understanding ownership)*
Remember that in Rust we have three kinds of types:

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

Do the following programs compile? Why (not)?

(a)

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

(b)

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

(c)

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

(d)

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

(e)

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

(f)

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

Exercise τH.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?

Exercise τH.8 *(Back to the roots)*

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

```rust
use std::sync::mpsc::{channel, Receiver, Sender};
use std::thread::spawn;

fn printer(rx: Receiver<u32>) {
    while let Ok(v) = rx.recv() {
        if v == u32::MAX {
            return;
        }
        println!("{v}");
    }
}

fn sieve(n: u32, prev_rx: Receiver<u32>, print_tx: Sender<u32>) {
    let (next_tx, next_rx) = channel();

    while let Ok(v) = prev_rx.recv() {
        if v == u32::MAX {
            match print_tx.send(u32::MAX) {
                Ok(_) => {
                    //do nothing
                }
                Err(e) => {
                    panic!("Not possible to send! Error: {}", e);
                }
            }
            return;
        }
        if v % n != 0 {
            print_tx.send(v).unwrap();
            spawn(move || sieve(v, next_rx, print_tx));
            break;
        }
    }

    while let Ok(v) = prev_rx.recv() {
        if v == u32::MAX {
            next_tx.send(u32::MAX).unwrap();
            return;
        }
```

```
        if v % n != 0 {
            next_tx.send(v).unwrap();
        }
    }
}

fn main() {
    let (tx0, rx0) = channel();
    let (print_tx, print_rx) = channel();
    spawn(move || sieve(2, rx0, print_tx));
    let p = spawn(move || printer(print_rx));

    for i in 2..1000 {
        tx0.send(i).unwrap();
    }
    tx0.send(u32::MAX).unwrap();

    p.join().unwrap();
    println!("finished!");
}
```

Exercise τH.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)` ans `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.

(b) Success and Failure are expected.

(c) Success is expected and failure is the exception.

(d) Absolutely unexpected error.

Exercise τH.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`

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

```
fn divide(numerator: f64, denominator: f64) ->
    ↳ Option<f64> {
    if denominator == 0.0 {
        None
    } else {
        Some(numerator / denominator)
    }
}
```

```
// The return value of the function
    ↳ is an option
let result = divide(2.0, 3.0);
// Pattern match to retrieve the
    ↳ value
match result {
    //?
}
```

(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.

```
fn no_42_allowed(argument: usize) -> Result<usize, &'static str> {
    if argument == 42 {
        Err("This cannot be the answer!")
    } else {
        Ok(argument)
    }
}
let example: usize = 42;
match no_42_allowed(example) {
    //?
}
```