Realms:

Lock-free Object Access in Multi-threaded Execution Environments

20. September 2022

Manuel Fischer, Christian Eichler, Henriette Hofmeier, Timo Hönig
manuel.fischer@rub.de

Bochum Operating Systems and System Software (BOSS)
Ruhr-Universität Bochum

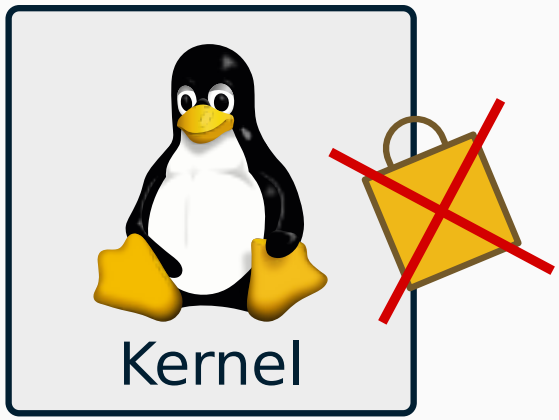
RUHR
UNIVERSITÄT
BOCHUM

RUB

Introduction: Thread Safety Validation

Problem: Thread safety validation at runtime difficult

Usage Scenario: porting sequencial code to multithreaded code



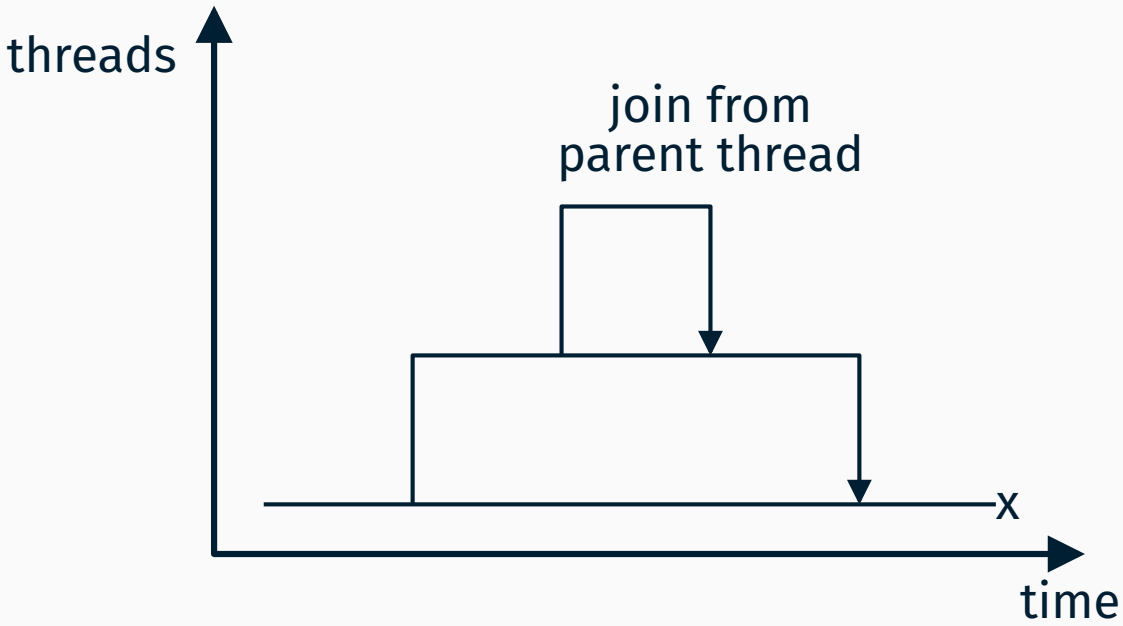
Goals:
- Lock free object access
- Minimal/no synchronization
- Lightweight thread safety validation at runtime
- Tool to reliable detect access conflicts

Approach: Thread safety system based on new concept **realms**
Zero cost abstraction: overhead only for debugging

Table of Contents

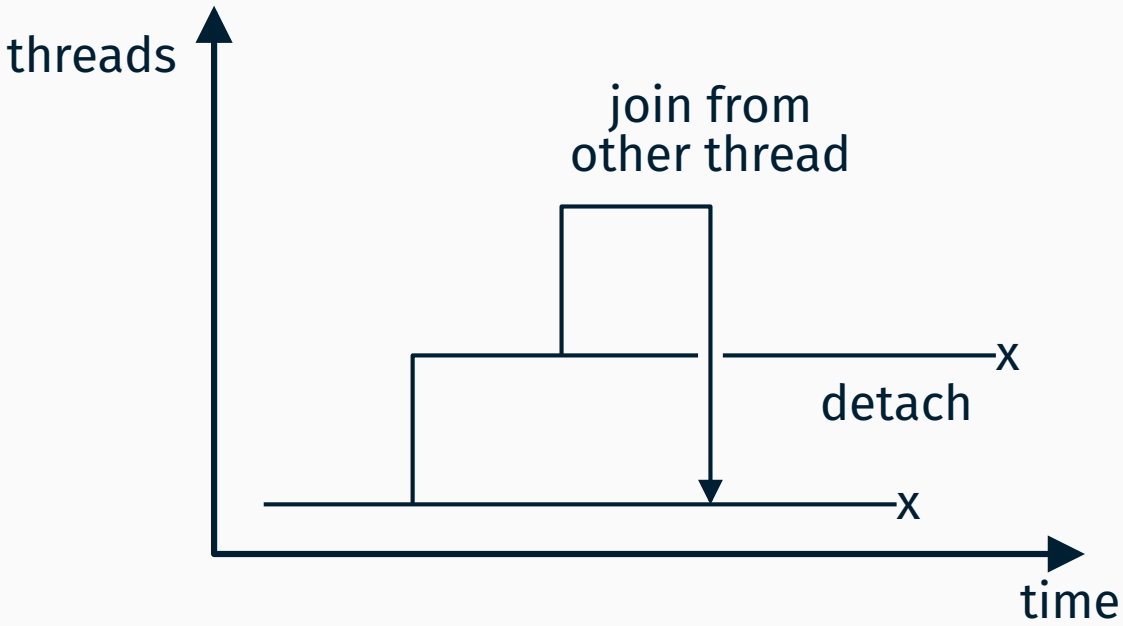
Threading Model

Hierarchical Thread Execution



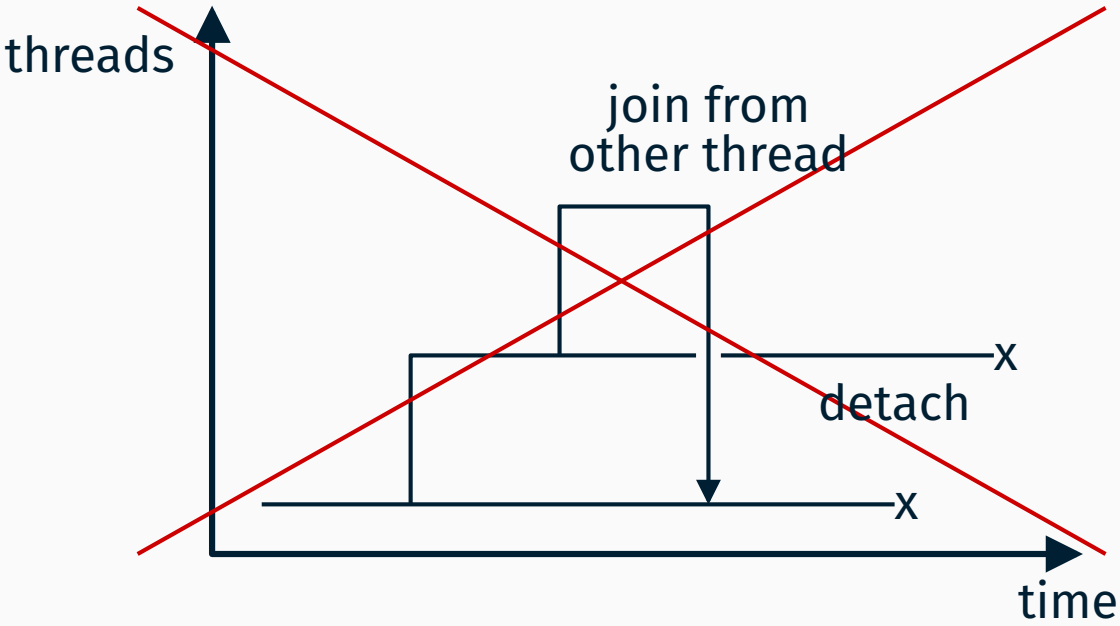
Hierarchical thread execution
child threads terminate before parents

Hierarchical Thread Execution



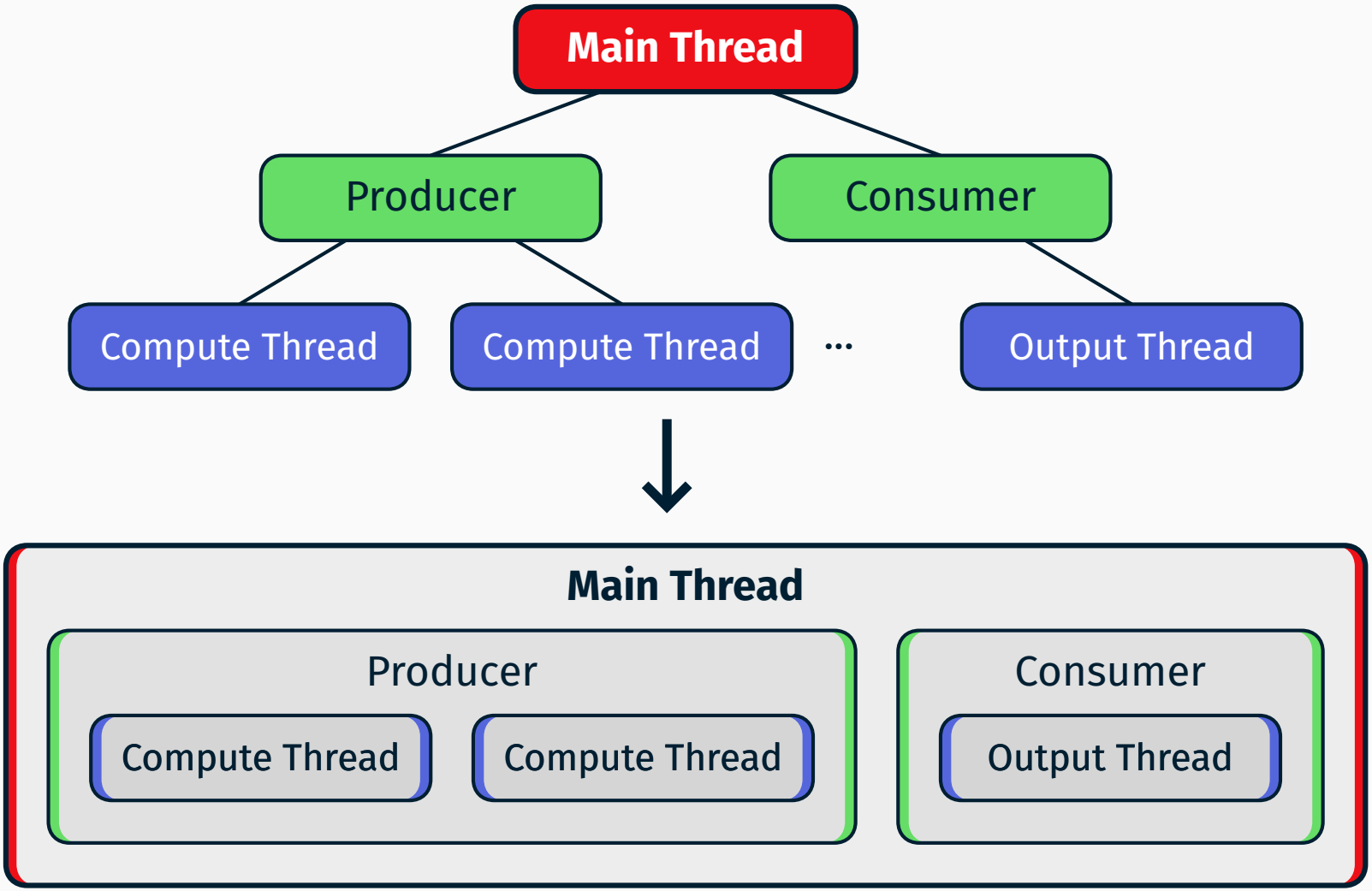
Problematic thread execution
overlapping lifetimes: dangling pointers

Hierarchical Thread Execution



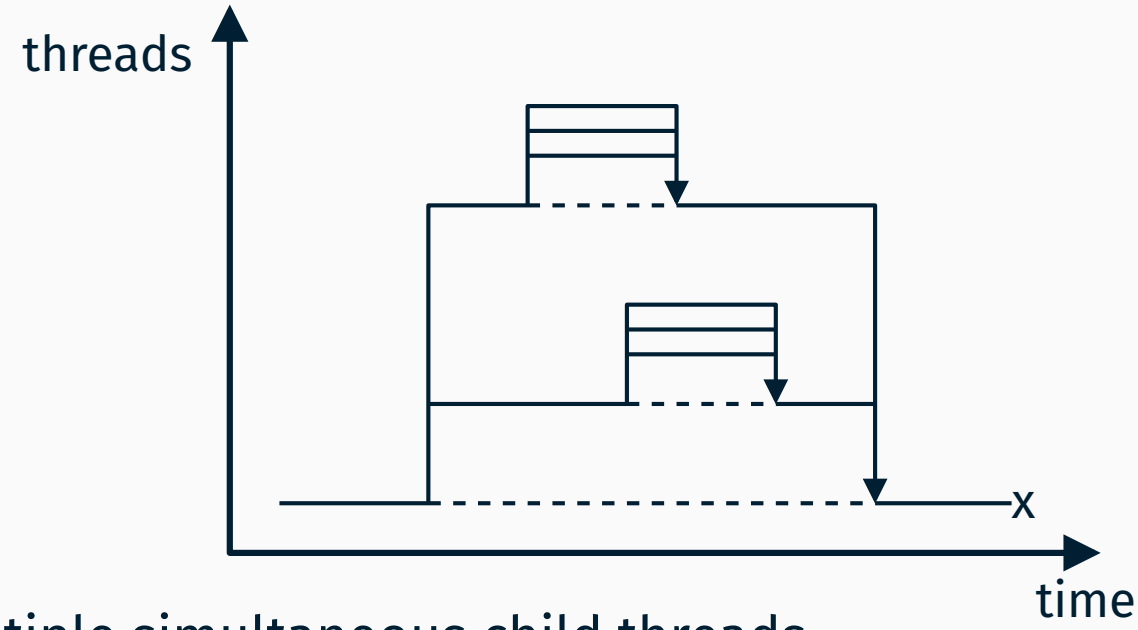
Problematic thread execution
overlapping lifetimes: dangling pointers

Threading Model



Consider child threads as parts of parent threads

Functional Thread Execution



- Multiple simultaneous child threads
- Parent thread waiting/no side effects on its realm
- Only side effect of child threads: computing a result
- Result: value, object graph or transaction
- Result processed in parent thread after all parallel threads completed

Realms

Realms

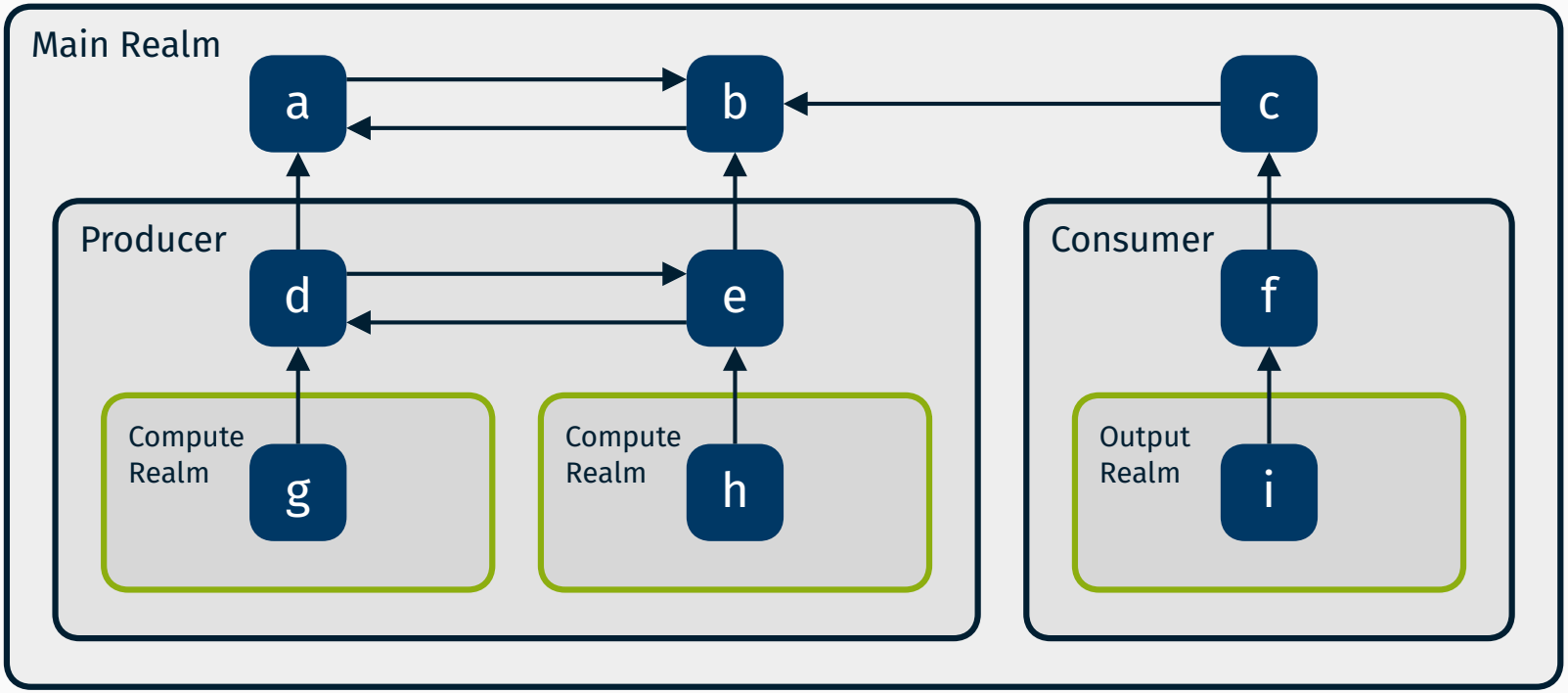
No modifications of unsynchronized objects can escape a realm

What? **Set of objects** with same ‘origin’ (thread or function call)
Restricted environment

Where? Multi threading, restricted code execution,
compile time bytecode evaluation

Why? Detect & prevent modifying objects associated
to a realm that is not the currently executed one

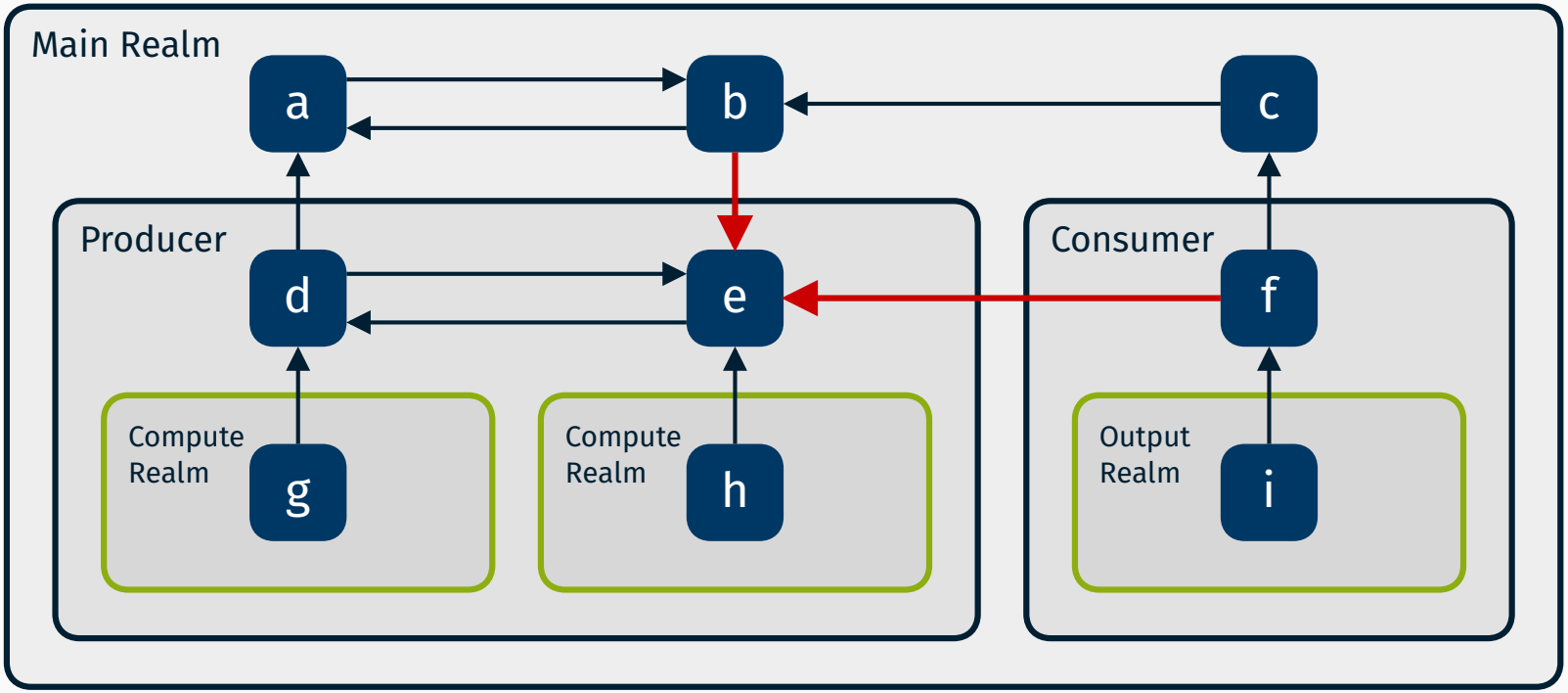
Hierarchical Realms: Object Graphs



Objects referring to other objects in other realms

Leaf realms: active realms

Hierarchical Realms: Object Graphs



Invalid: References to child realm and cross references

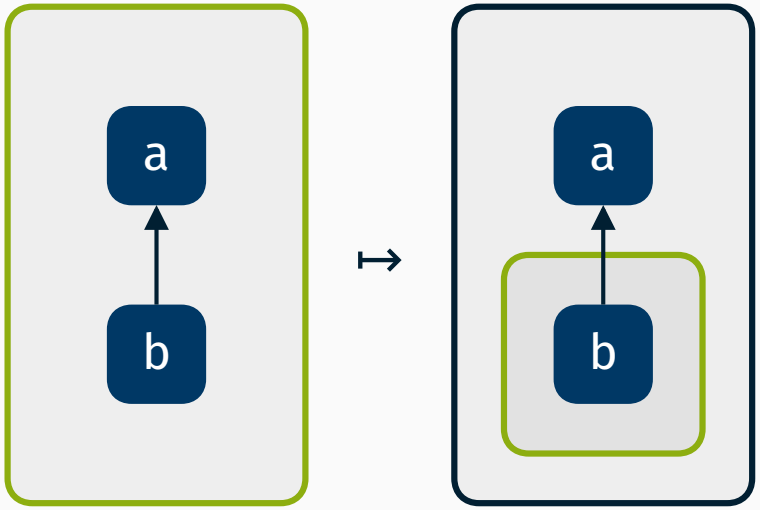
Impossible if only **leaf realms** are modifyable

Realm Operations

Splitting Realms

Why?

- Create child realm
- Modify existing objects in place
- Split work among threads



When?

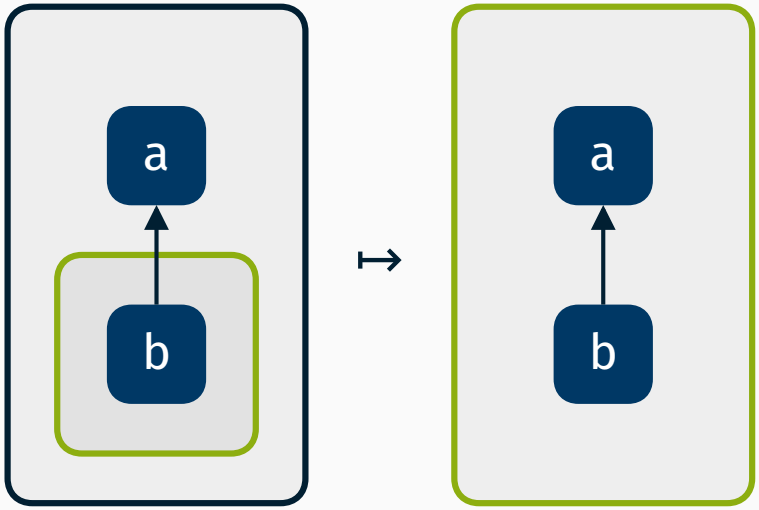
- Threads: Before creation of threads
- Function call: Before function call

How?

- Reassociate objects with new realm
- Ensure objects in new realm are not reachable from original realm anymore
  1. Keep objects in realm or copy objects, if still reachable
  2. Hide or remove references to objects moved to the child real

Resolving Realms

Splitting realms ↔ resolving realms



Why?

- Continue parent realm
- References to child realm from parent realm

When?

- Threads: after join in parent thread
- Function call: after function returned

How?

1. Reassociate objects with parent realm
2. Update reference counters of parent realm objects referred to from child realm (Only count references inside a realm: → *no synchronization of reference counter*)

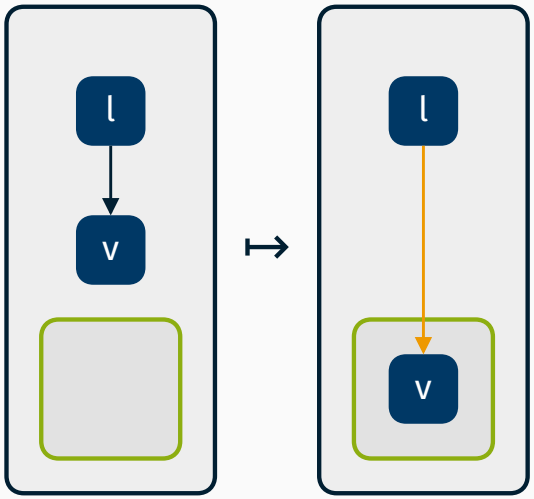
Synchronization

Where? Queues, synchronized variables (l)

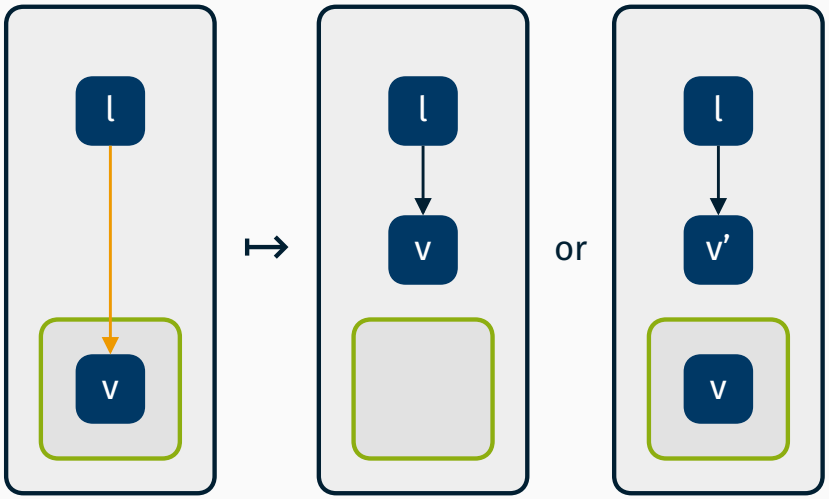
Why? Shared mutable state, communication between threads

How? Reassociate object graph (v) with other realm
consider references into v

Acquire: Queue pop, mutex lock



Release: Queue push, mutex unlock



no copy
if no other
reference to v

deep copy
if multiple
references to v

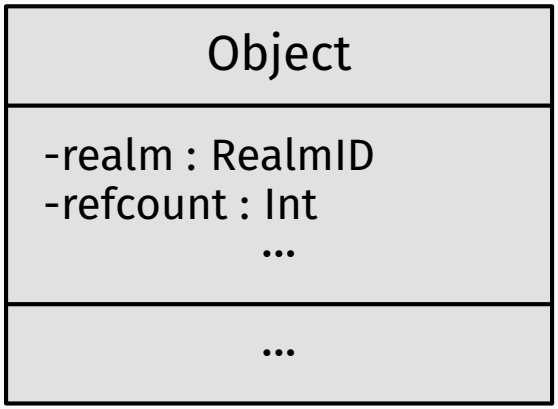
inaccessible locked reference

unlocked state: objects in v/v' only reachable through l

Implementation

Implementation

```
thread_local RealmID current_realm;

shared Object* alloc_object()
{
    Object* p = malloc(...);
    p->realm = current_realm;
    return p;
}

const Object* get_read(shared Object* p) {
    return p;
}

Object* get_write(shared Object* p) {
    assert(p->realm == current_realm);
    return p;
}
```

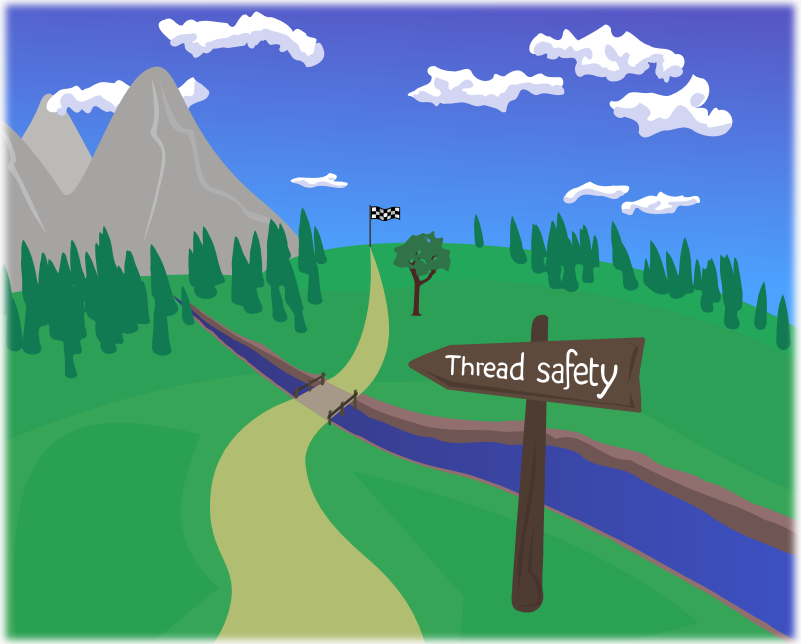


Example Error Cases

```
void my_thread(shared queue* q) {
    shared element* e = pop_synchronized(q);
    destroy_queue(get_write(q)); // runtime error

    get_write(e)->name = "Alice"; // ok

    begin_realm(); // could split realm here
        puts(get_read(e)->name); // ok
        get_write(e)->name = "Bob"; // runtime error
        destroy_element(get_write(e)); // runtime error
    end_realm();

    get_write(e)->name = "Charlie"; // use after free
    // destruction of e should happen here
}
```

Conclusion

Conclusion



Goals: Strict runtime thread safety validation using realms

Approach: Realms as optional light debugging or safety feature
Avoiding data races by hierarchical threading model

Result: Many access conflicts detectable with realms
Not just useful for multi-threading: restricted execution

Implementation: `gitlab.rub.de/realms/realms-cpp`

Appendix

Example Pseudocode: Output Thread

```
void consumer_thread(shared output_context* args) {
   begin_realm(); // consumer realm
      int output_flags = get_read(args)->output_flags;
      get_write(args)->output_flags |= 2; // error

      shared result* r;
      while(r = pop_result(ctx->queue)) {
         get_write(r)->sum /= get_read(r)->count; // ok

         begin_realm(); // output realm
            ...
            destroy_result(get_write(r)); // error
         end_realm();
         // destruction should happen here instead
      }
   end_realm();
}
```

Example Pseudocode: Output Thread

```
struct output_context {
    realm_id realm;

    int output_flags;
    shared result_queue* queue;
};

struct result {
    realm_id realm;

    int count;
    double sum;
};

void destroy_result(result* r);
```

Garbage Collection

Usual Problem:	Synchronization of reference counter
Solution:	Only count references inside own realm, excluding child realms
Explanation:	Possible because if child realms refer to an object, a reference must already exist in the realm, used to get a reference to the object
Result:	**No locking necessary** Generally: any GC-algorithm per realm → more efficient

Reference Counting

```
void inc_ref(shared Object* obj) {
    if(!obj) return; // null guard
    if(obj->realm != current_realm) return; // realm guard

    obj->refcount++;
}

void dec_ref(shared Object* obj) {
    if(!obj) return; // null guard
    if(obj->realm != current_realm) return; // realm guard

    obj->refcount--;
    if(obj->refcount == 0) destroy_object(obj);
}
```

Realm Creation

```
void begin_realm() {
    current_realm++;
}

void end_realm() {
    current_realm--;
}
```

4 Phase Thread Execution

Enable simultaneous thread execution:

1. Argument allocation, realm creation
2. Thread starting, realm activation
3. Thread joining
4. Result collection, realm resolution