Managarm: A Fully Asynchronous Operating System Powered By Modern C++

Alexander van der Grinten
avdgrinten@managarm.org

The Managarm Project

Bio

Postdoctoral researcher, CS

- Humboldt-University of Berlin, Germany
- PhD in 2018 from University of Cologne, Germany

Research focus: engineering of parallel algorithms, graph algorithms, hard combinatorial problems

This talk: Managarm, a pragmatic OS with fully asynchronous I/O

- Open-source project
- Founded in 2014
- Many active contributors

Agenda

- Why does async matter?
- Managarm: a fully asynchronous OS
  - Brief overview
  - Async in C++20: high-level code and building blocks
  - Implementation of async system calls in C++20
- Open challenges and conclusions

Synchronous I/O

Example: POSIX files + OpenGL: straightforward, well-understood

```
void loadTexture(const char *path) {
    char buffer[4096];
    int fd = open(path, O_RDONLY);
    ssize_t n;
    while ((n = read(fd, buffer, 4096)) > 0) { // Read data.
        glTexSubImage2D(/* ... */, buffer); // Upload to GPU.
    }
}
```

... but does not scale well.

- `open()`, `read()`, `glTexSubImage2D()` block the current thread.
- What if thousands of files need to be read? We cannot start 1000 threads.

... and does not map well to modern hardware.

Modern hardware is highly asynchronous

Evolution of storage technology:

SATA (= AHCI), 2004:	1 request queue	32 reqs/queue
NVMe, 2011:	65535 request queues	65535 reqs/queue

When using blocking I/O, a thread posts **one request** to **one queue**.

Similar story for:

- GPUs
- Networking (ethernet, WiFi, Infiniband, ...)
- Accelerators / offloading (AI, crypto, ...)
- Audio
- USB / Thunderbolt

= almost every device in your PC / server!

How can we do it better in C++20?

```
async::task<void> loadTexture(const char *path) {
    char buffer[4096];
    int fd = co_await posix::open(path, O_RDONLY);
    ssize_t n;
    while ((n = co_await posix::read(fd, buffer, 4096)) > 0) {
        co_await ogl::glTexSubImage2D(/* ... */, buffer);
    }
}
```

"But that is not how it works!"

- "Why not?" – "The OS + libraries do not work that way!"

NB: there is `io_uring`, Vulkan, . . ., but no general solution.
Even `io_uring` often falls back to (kernel-)thread-based emulation.

Managarm: A fully asynchronous OS

Managarm is an operating system that was designed to be asynchronous from the ground up.

Everything is asynchronous, notably:

- file system I/O (including metadata updates),
- hardware drivers (`co_await` instead of callbacks and interrupts),
- device operations (`ioctl`),
- memory management

and also: graphics, networking, ... as in other OSes.

Managarm: high-level overview

Managarm employs a microkernel:
drivers run as "normal" C++ programs, without supervisor privileges.

Kernel (= supervisor mode):

- Memory management
- Scheduling
- IPC

Written in freestanding subset of C++20,
STL-like support library,
custom memory allocation.

User mode:

- Hardware drivers
- POSIX emulation
- Applications

Written in standard C++20,
access to full STL.

Managarm: features and status

Good Linux source-level compatibility:

- `epoll`, `eventfd`, `timerfd`, `signalfd`, ...
- Linux-like `/dev`, `/proc`, `/sys`.
- Runs Wayland + X11 desktop apps.
- Translation to (blocking) Linux API happens in libc (which is written in C++ ;).

Hardware support:

- Lots of virtualized hardware.
- Real hardware: WIP, many modern devices.

Current status: functional but not stable yet.

Case study: async in low-level driver code

Recall: hardware is asynchronous already.

Usual flow of a driver (e.g. to read from disk):

1. Driver writes commands to a buffer in RAM.
2. Driver notifies device.
3. Device reads commands from RAM and performs work.
4. Device notifies driver using an interrupt.
   Interrupt = "hardware signal handler",
   causes CPU to jump to another function.

- Traditionally, interrupts are handled by callbacks.
- In Managarm, we can use the full power of async C++20 to handle them.
- It becomes rather easy to write *correct* and *concurrent* drivers.

... but let us look at the traditional code first.

Handling interrupts via callbacks (the painful way)

```
std::mutex mutex;
std::deque q;

void interrupt_callback() { // Assumption: not called in re-entrant way.
    // Have to lock: do not know which thread we interrupt!
    std::lock_guard lock{mutex};
    q.push_back(dev.get_byte()); // Fetch data from device.
}
// Hypothetical driver code: needs to parse uint16_t words from device.
void parse_bytes() { // Do work here to avoid interrupt context.
    disable_interrupts();
    {
        std::lock_guard lock{mutex};
        if (q.size() < 2) // (More or less) explicit state machine here.
            return; // Retry when we are called again.
        auto b0 = q.pop_front();
        auto b1 = q.pop_front();
        auto word = (b1 << 8) | b0; // Combine to uint16_t.
        [...] // Process word from device.
    }
    enable_interrupts();
}
```

Handling interrupts with coroutines in C++20

```
async::queue<uint8_t> q; // Note: data structure is now async.

async::task<void> handle_interrupts() {
    uint64_t sequence = 0;
    while(true) {
        auto evt = co_await helix::awaitEvent(interrupt, sequence);
        q.push_back(dev.get_byte()); // Fetch data from device.
        sequence = evt.sequence();
    }
}
async::task<void> parse_bytes() {
    while(true) {
        auto b1 = co_await q.pop_back();
        auto b2 = co_await q.pop_back();
        auto word = (b2 << 8) | b1; // Combine to uint16_t.
        [...] // Process word from device.
    }
}
```

Coroutines allow us to: (i) write more concise code, (ii) avoid the need for explicit state machines and (iii) remove complex locking.

[Questions?]

Async syscalls in Managarm

```
co_await helix::awaitEvent(interrupt, sequence);
```

`helix::awaitEvent()` is an asynchronous syscall in Managarm.
syscall = call from user mode into kernel, e.g. `open()`, `read()`, … in Linux.

Other async syscalls for: IPC, task management, memory management, …

Let us take a closer look at this mechanism:

- Syscall is invoked by user mode.
  Stub: `helSubmitAwaitEvent(HelHandle interrupt, uint64_t sequence, HelHandle ringBuffer, void *context)`
- Syscall performs some async work in the kernel,
  i.e. waiting until an interrupt happens.
- Kernel notifies user mode after completion of syscall.

Two layers involved in this async syscall: (i) the OS and (ii) C++.

Notification mechanisms for async operations

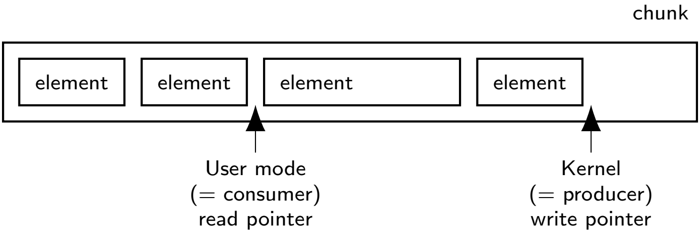
Kernel needs some mechanism to notify user mode that async work is done.

- Linux has `epoll`, or newer: `io_uring`.
- Windows has I/O completion ports (IOCP).

In Managarm: use lock-free ring buffer for kernel → user mode notification.
Similar idea now also in Linux/other OSes, e.g. as `io_uring`.

- Zero switches from kernel ↔ user mode on the fast path.
- Managarm's only blocking syscall: block on atomic variable
- Corresponds to `std::atomic<unsigned int>::wait()` in C++20
  = futex in Linux or `WaitOnAddress` in Windows.

Async syscalls: the OS layer




```
struct HelChunk {
    // Kernel stores its write pointer after producing elements.
    // User mode blocks via std::atomic's wait() method (C++20).
    std::atomic<unsigned int> progress_futex;
};
struct HelElement {
    unsigned int length; // Length of the element in bytes.
    void *context; // User-defined value.
    ... // Syscall-specific results follow here.
};
```

Async syscalls: the C++ layer

On the C++ side: need some representation for async primitives like `helix::awaitEvent()`.

What about `awaitable`?

```
template<typename A>
concept awaitable // Makes A co_await-able (simplified).
= requires(A a, std::coroutine_handle<> h) {
    a.await_ready();
    a.await_suspend(h);
    a.await_resume();
};
```

Need coroutine (= `std::coroutine_handle<>`) to wait for completion of `awaitable` operations.

- `awaitable` as primitive ⇒ every consumer of async operations (e.g. every async algorithm) needs to be a coroutine.

Awaitable as a primitive

Coroutines work well for high-level logic.

Rule of thumb: 90% of your async C++20 code will be coroutines.

But coroutines have some overheads. Consider:

```
async::task<std::optional<T>> pop_back(cancellation_token ct);

async::task<T> pop_back_nocancel() {
    auto r = co_await pop_back(cancellation_token{});
    co_return std::move(*r);
}
```

Calls to coroutines potentially allocate coroutine frame.

- Not desirable for building blocks / every async algorithm.
- Hence: `awaitable` is not the right primitive.

Senders/Receivers for low-level code

Borrow sender/receiver concepts of current executors proposal
(not related to networking, despite the name).

Disclaimer: what you see here is an indepedent implementation, not a standards proposal.

(Credit for ideas goes to SG1, credit for bugs goes to me.)

Two concepts per async operations, informal definitions:

- `sender`: class that knows how to initiate an async operation (= tuple of arguments)
- `operation`: class that represents state of async operation

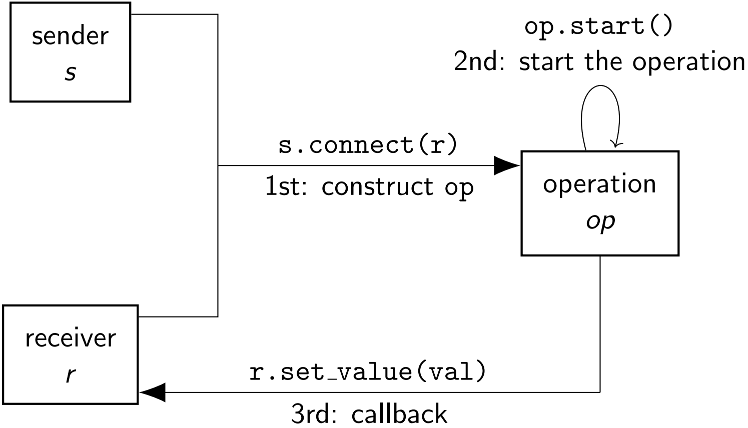
One concept for consumers of async operations:

- `receiver`: class that knows how to resume after async operation completes (= callback)

More details: Cppcon'19 talk by David Hollman, Eric Niebler.

Senders/receivers: brief explanation

Senders/receivers proceed in three steps:



Async algorithms

Senders/receivers enable zero-overhead async algorithms:

```
async::task<std::optional<T>> pop_back(cancellation_token ct);
auto pop_back_nocancel() {
    return async::transform(pop_back(cancellation_token{}),
        [] (optional<T> &&r) { return std::move(*r); });
}
```

Examples of async algorithms:

- `async::transform()`
- `async::sequence()`
- `async::race_and_cancel()`

We also make heavy use of async data structures:

- `async::queue`
- `async::recurring_event`

[Questions?]

Async syscalls: sender/receiver implementation

Let us try to write `sender` and `operation` implementations for `helix::awaitEvent()`.

Some general boilerplate:

```
// Callback that is invoked from ring buffer code.
struct AsyncSyscall {
    AsyncSyscall(const AsyncSyscall &) = delete;

    AsyncSyscall &operator= (const AsyncSyscall &) = delete;

    virtual void notify(HelElement *elem) = 0;
};
// Data type to represent the result of helix::awaitEvent().
struct AwaitEventResult {
    AwaitEventResult(HelElement *elem);

    [...] // More functions, members, etc.
};
```

Async syscalls: implementation of operation

```
template<typename R>
struct AwaitEventOperation : AsyncSyscall {
    HelHandle interrupt;
    uint64_t sequence;
    HelHandle ringBuffer;
    R receiver;

    void start() {
        helSubmitAwaitEvent(interrupt, sequence, ringBuffer, this);
    }

    void notify(HelElement *elem) override {
        execution::set_value(receiver, AwaitEventResult{elem});
    }
};
```

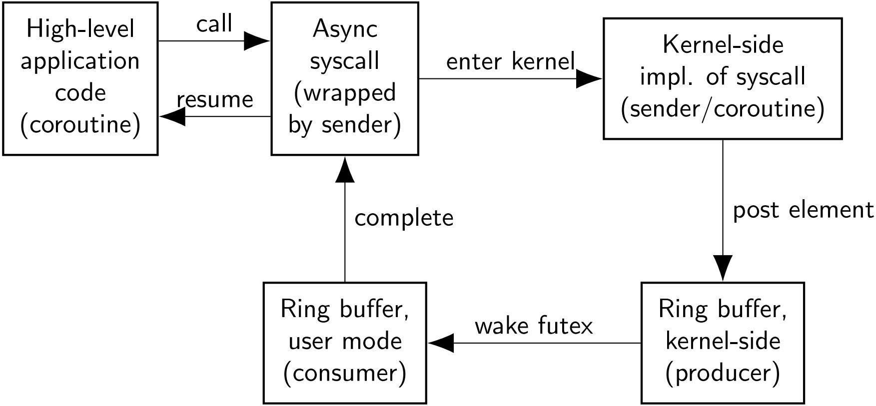
Async syscalls: sender boilerplate

```
struct AwaitEventSender {
    HelHandle interrupt;
    uint64_t sequence;
    HelHandle ringBuffer;

    template<typename R>
    AwaitEventOperaton connect(R receiver) {
        return {interrupt, sequence, ringBuffer, std::move(receiver)};
    }
};

AwaitEventSender awaitEvent(HelHandle interrupt, uint64_t sequence,
                            HelHandle ringBuffer) {
    return {interrupt, sequence, ringBuffer};
}
```

Async syscalls: the full picture



Challenge: synchronous fast paths for senders/receivers

Recall: we use a modified version of senders/receivers

Most important change: synchronous fast paths

- Many operations only need to perform I/O on slow paths (e.g. caches that fetch from disk).
- For these: do not want to invoke a callback on completion (otherwise stack frames accumulate, especially in loops).

Our solution: add a function `pass_value` to receivers.

- Passes result to receiver, but does not cause receiver to resume control flow.
- Control flow resumes synchronously after `pass_value()`.
- Solution within the executors proposal would be desirable.

Challenge: error handling

C++20 solved async for our use case. What is next?

Error handling quite messy for idiomatic low-level C++ code.

- In Managarm: switch to `expected<T>`.

Would like to use exceptions (e.g.for errors in RAII ctors)

- But: non-deterministic behavior not acceptable (side channels, real-time latencies)
  $\Rightarrow$ primary problem: predictability, not performance.
- Deterministic exceptions would solve this issue.

Conclusions

Mainstream OSes do not handle async very well.

- Few operations are entirely async;
  common fallbacks to blocking code
- But: designing for async from the group up is necessary to fully utilize modern hardware

C++20 is well-suited for async low-level code. Main tools:

- Coroutines for high-level code
- Senders/receivers as building blocks (zero-overhead abstraction)
- Async algorithms to compose building blocks

Given these tools, we can write async code by default.

- Writing correct concurrent drivers becomes easier

Acknowledgements

Check out the project: `github.com/managarm/managarm`

Blog: `managarm.org` Twitter: `@managarm_OS`

Thanks to all contributors!

Co-owner of the project: Kacper Słomiński (@qookei).

Contributors: Arsen Arsenović (@ArsenArsen), Dennis Bonke (@Dennisbonke), Geert Custers (@Geertiebear), Matteo Semenzato (@Matt8898), Thomas Woertman (@thomtl), @0xqf, @aurelian2, @Itay2805, @itsmevjnk, @fido2020, @Menotdan, @mintsuki, @no92, @PositronTheory, @RicardoLuis0, @streaksu, and @toor.