ChatGPT

Rust Generators (`gen` Blocks) vs. `may` Coroutines for Long-Running I/O on ZFS

Context: Read-Heavy ELT Pipelines with S3 and ZFS Cache

In a big-data ELT pipeline, remote files (e.g. on Amazon S3) are often cached to a local storage (like a ZFS pool) to improve subsequent read performance. The challenge is to stream large files from S3 to disk and to the processing pipeline efficiently, without loading everything in memory or stalling on I/O. We want to avoid heavyweight async frameworks (Tokio/async-std), instead exploring **synchronous generators** (Rust's upcoming `gen` blocks) and the `may` **coroutine** library. Key considerations include I/O throughput, reliable handling of blocking operations, backpressure (preventing producers from outrunning consumers), and any quirks of using Rust on ZFS (like long sequential reads and prefetch behavior).

Synchronous `gen` Blocks for Disk I/O Streaming

Rust's **synchronous generators** (proposed via `gen { ... }` blocks) let you write an iterator as a simple function that `yield`s values, rather than manually implementing the `Iterator` trait. This is much more ergonomic for streaming data. As the RFC notes, **"with `gen` blocks, we can now write a simple `for` loop and still get a lazy iterator of values"**, avoiding the pain of manual iterator state machines [1]. In practice, a `gen` block can wrap a file-reading loop: read a chunk from disk, `yield` the chunk, and repeat until EOF. Each `yield` suspends the generator, producing a value for the caller, and resumes on the next iteration call.

Performance: Synchronous generators are essentially **compile-time transformed state machines**, so they add minimal overhead on top of the raw I/O calls [2]. Unlike thread-based approaches, they don't allocate a separate stack per task or incur heavy context switches. Each chunk read is produced on-demand, which is memory-efficient. As Kevin Hoffman notes, iterators (and by extension generators) are *lazy* – they produce data "only when asked," so you **never load more than a chunk or two in memory at once** [3]. This laziness inherently provides **backpressure**: the reading generator will block (pause) until the consumer pulls the next item. Downstream processing can be done in normal iterator combinators (e.g. mapping or filtering each chunk), which keeps the code simple and idiomatic. For example, one could iterate over a chunk generator and compute hashes or parse records, all in a streaming fashion, without ever materializing the whole dataset at once.

Example: With `gen`, your disk reader might look like:

```
fn read_chunks(path: &Path, chunk_size: usize) -> impl Iterator<Item=Vec<u8>> {
    gen!({  // hypothetically using a gen block
        let mut file = File::open(path).expect("open failed");
        let mut buffer = vec![0; chunk_size];
```

```
        loop {
            let n = file.read(&mut buffer)?;
            if n == 0 { break; }           // EOF
            yield buffer[..n].to_vec();    // yield an owned chunk
        }
    })
}
```

This yields `Vec<u8>` chunks which the caller can consume in a `for` loop or iterator chain. The above is synchronous (the `read` call will block the current thread until data is available), but because it's an iterator, it won't progress faster than the consumer asks for data, naturally preventing overflow. The upcoming Rust generator design integrates directly with the iterator API – a `gen { ... }` expression produces an object you can directly `.into_iter()` and use in a `for` loop [4]. This means you can easily compose it with other iterators or use it in streaming pipelines. For instance, you could do:

```
for chunk in read_chunks(file_path, 1<<20) {
    process(chunk);
}
```

or even `read_chunks(...).map(process).filter(...).collect()`, etc., leveraging high-level iterator adapters.

Considerations: One limitation of the current iterator API is that it doesn't allow holding a yield across a mutable borrow without unsafe code [5] [6]. In practice, that means if you want to **avoid copying** and yield a *borrowed* slice of a single buffer, it's tricky – you'd likely need to yield owned buffers (as in the example above, using `to_vec()`). This incurs an allocation and copy per chunk. However, for large chunk sizes the overhead of an allocation per few MB is usually negligible compared to disk I/O cost. If zero-copy is required, more complex patterns or unsafe code would be needed. But generally, using owned chunk buffers is simpler and quite efficient. The generator itself is stackless and lightweight – it doesn't spawn threads or use an async runtime; it just turns your loop into an iterator. This approach is ideal if your pipeline is mostly synchronous: e.g. reading from a fast local cache on ZFS and processing in the same thread. It maximizes data locality and leverages Rust's strong optimizations for iterators.

`may::go!` Coroutines for Overlapping Network & Disk I/O

While `gen` works great for **local** I/O streaming, reading from **remote S3** and simultaneously writing to disk might benefit from concurrency to hide latencies. `may` is a crate that provides **stackful coroutines** (think Go-like "green threads") in Rust [7]. With `may`, you can spawn lightweight tasks using `may::go!` that behave like threads but are scheduled cooperatively on a pool of OS threads. Crucially, `may` **makes I/O operations appear blocking without actually blocking the OS thread**: *"With May, you can write your code as if it were synchronous… as if I/O calls were blocking, except in reality they won't block – when they would block, something else will be scheduled."* [8] In other words, `may` intercepts or uses non-blocking I/O under the hood, so a coroutine performing a blocking operation (e.g. waiting on a socket or file descriptor) will yield control, allowing other coroutines to run. This is done via an internal I/O scheduler (using epoll, IOCP, etc., rather than Tokio's `mio`) [9].

Using `may` for S3 and Disk: We could spawn one coroutine to handle the S3 download and another to handle writing to ZFS (or even reading-back for processing). For example, one coroutine could perform an HTTP GET to S3, reading data in chunks, and push those chunks into a thread-safe queue or channel. Another coroutine could pop from the queue and write to the local ZFS file, or process the data concurrently. Because `may` scheduling is cooperative, when the S3 coroutine waits for network data, the disk-writing coroutine can run, and vice versa. This overlap can significantly improve throughput in high-latency or high-bandwidth scenarios. For instance, if writing to disk is momentarily slower than reading from S3 (maybe due to bursty disk flushes), the S3 task will naturally pause (its send/queue operation will block when the buffer is full), allowing the writer task to catch up – this is **backpressure via blocking**. Conversely, if the network is the bottleneck, the disk task will mostly wait idle; `may` can then schedule other tasks (maybe other file transfers) in the meantime.

Backpressure & Coordination: In a `may` setup, you typically use synchronization primitives like channels (queues) or semaphores to coordinate tasks. For example, an **MPMC channel** (which `may` provides) can be bounded to, say, a few chunks capacity. The producer coroutine (S3 reader) `send`s chunks into it, and the consumer (disk writer or processor) `recv`s from it. If the channel is full, the producer's send will yield until the consumer catches up – effectively **throttling the download** so as not to overflow memory. This approach ensures you don't pull data from S3 faster than you can write to disk. It's similar to the backpressure behavior of iterators, but in an asynchronous, push-based form. The key difference is that here the coroutines run *concurrently* (on possibly different CPU cores if you configure `may` to use multiple threads [10]). This can **maximize throughput** by utilizing CPU and I/O in parallel: e.g. one coroutine computing or compressing data while another waits for I/O. In contrast, a single-threaded generator/iterator would do these steps sequentially.

Performance: Each `may` coroutine carries its own stack and context. Context switching a stackful coroutine is generally a bit heavier than moving between states in a compiled state machine (like an async Future or a generator) [2]. There's some memory overhead per coroutine (the stack size, which is configurable) and a cost to switch stacks. In practice, for I/O-bound tasks, this overhead is small – the time spent waiting on network/disk far outweighs a few nanoseconds switching coroutines. In fact, real-world benchmarks have shown that a stackful coroutine approach like `may` can achieve performance on par with Tokio's async/await in many scenarios [11]. One of the TechEmpower benchmark top-performers was built on `may` (the **may-minihttp** server), demonstrating that if used correctly, `may` can be extremely fast [12]. The benefit of `may` is mainly **productivity and scaling**: you can spawn thousands of coroutines (similar to goroutines) without spawning thousands of OS threads, and write simpler code than managing a thread pool manually. Each coroutine can block on I/O in code, but **non-blocking underneath** means you can run *many* concurrent I/O tasks with high throughput.

Reliability and Caveats: Because `may` operates outside the standard async ecosystem, there are some considerations. You must use `may`'s versions of blocking operations (e.g. its networking types) or ensure file descriptors are in non-blocking mode. As one user noted, you often need "specialized primitives (e.g. a different TCP stream than the one from `std`)" to work seamlessly with `may` [13]. For S3, this might mean using a crate or HTTP client that can use a custom socket or doing the HTTP request manually using `may::net` sockets. If a library internally uses normal blocking calls (and `may` doesn't patch those), it could block the entire OS thread. One strategy is to perform the S3 HTTP GET in a separate OS thread if using a blocking client, but ideally one would use `may`-friendly I/O throughout. Another consideration: `may`'s coroutine spawning is marked `unsafe` in the API because of low-level tricks (it uses assembly to swap stacks, etc.) [14]. This is generally safe if you stick to pure Rust code without thread-local assumptions, but it *can* lead to undefined behavior if, for example, a library call uses thread-local storage (TLS) that isn't expecting to be migrated across coroutine yields [15]. In practice, many have used `may` successfully, but it's a less "official" path and requires discipline (you essentially run your whole pipeline inside `may` to avoid mixing with standard futures or OS-thread-based APIs).

Summary of `may` usage: It shines when you have **multiple I/O-bound tasks** that can be overlapped. For our scenario, if you need to download, cache, and maybe process *many files concurrently*, `may` could handle that with a small thread pool and many coroutines, whereas spawning a dedicated OS thread per file could become heavy. If you're only dealing with one large stream at a time, `may` is not strictly necessary – a single-thread generator + the OS's natural read-ahead might suffice. But if you want to **prefetch from S3 while processing and writing**, `may` gives a convenient structure for it (without rewriting your logic in terms of `Future`s and `async/await`). You get to write mostly straightforward code (loops, reads, writes) and let the `may` scheduler interleave them as needed.

ZFS Considerations: Long Reads and Prefetching

ZFS is known for its robust caching (ARC) and intelligent prefetch algorithms, which is great news for streaming large files. When you read a file sequentially on ZFS, the filesystem will detect the pattern and **proactively prefetch subsequent blocks into the ARC** (Adaptive Replacement Cache) ahead of your reads. In fact, ZFS often **aggregates read requests into larger chunks** than the application requests, to reduce disk IOPS overhead [16]. For example, if your code is reading 128KB at a time, ZFS might internally pull in a few MB more speculatively. This means that using a reasonably large chunk size (say 1 MB) in your Rust code will align well with ZFS's strategy – you'll effectively be reading at full disk bandwidth with minimal syscall overhead. A user on the ZFS forum noted that if you set the ZFS record size to 1MB (the maximum), *"you're kind of already prefetching"*, since each read will grab a big record from each disk [17]. In other words, a large record size or read size makes each I/O more efficient and sequential. Many ZFS users don't manually tweak prefetch beyond that, as ZFS's default prefetch works well for most streaming workloads [18].

For **long-lived sequential reads**, ZFS's prefetch can dramatically improve throughput – it will keep the disks busy by reading ahead of your application's read pointer. As long as your reading pattern is mostly linear, ZFS will continue to fetch the next chunks in advance. If your pipeline's consumer slows down (processing data), ZFS may fill some of its ARC (RAM) with prefetched data, which will simply wait there until your program issues the next read. There's no harm in that except using some memory (ARC will evict older data if needed). If the stream is truly one-pass and very large (far exceeding ARC size), ZFS will eventually evict chunks as you go (potentially using an LRU/ARC strategy). In such cases, ZFS has heuristics (like "sequential scan avoidance") to avoid trashing the cache with one-time reads – so it might not keep *all* data in ARC if it thinks you won't re-read it. This is good for not evicting other hot data. Essentially, ZFS is optimized for exactly this scenario of scanning large files.

Concurrency on ZFS: If you use `may` to read multiple files concurrently or to read and write at the same time, ZFS is quite capable of handling it. ZFS uses its I/O scheduler and multiple worker threads under the hood. Prefetch is per-dataset and per-file; concurrent sequential reads on different files will each get their own prefetch streams. Do note that if you turn one large sequential workload into a highly concurrent access pattern (e.g. by reading many files in small interleaved chunks), you could defeat prefetch (making the access pattern effectively random from the disk's perspective). However, if each coroutine deals with a different file sequentially, the disk heads (on HDD) might seek between files. On SSDs or multi-vdev pools, this is less of an issue. For spinning disks, one might prefer to stream one file at a time or use ZFS's QoS settings to avoid excessive seek thrashing. But given the context (ELT pipelines), it's common to have multiple files (partitions of a dataset) read in parallel – ZFS can handle that, though the throughput per file might drop compared to a single-stream case due to head seeking (if on HDD).

Writing to ZFS: Our caching layer writes the S3 data to ZFS. By default, ZFS will buffer writes in memory (ARC is mostly for reads, but it has a separate transaction log and write cache system). Unless mounted with special flags, writes will be asynchronous – your `write()` calls return quickly after copying data into ZFS's memory. Every few seconds, ZFS flushes those writes to disk in batches. This means your `may` coroutine writing to disk likely won't block much unless you absolutely saturate the write cache. However, if the cache (ARC) gets full or if you do a very large burst, the write call could block on ZFS I/O at times. In a pipeline, that just means the writing coroutine will yield, and the reading (S3) coroutine might pause if the channel is full – again ensuring backpressure so you don't outrun the disk. One ZFS-specific tip: if you *don't care about immediate durability* of the cache file (since it's just a copy of S3 data), you could mount the ZFS dataset with `sync=disabled` to avoid the small overhead of the ZIL (intent log) on each transaction [19]. This can boost write throughput significantly (at the risk of losing the last few seconds of writes on power loss, which might be acceptable for a cache). Even without that, ZFS can typically ingest sequential writes at disk speed, especially if the pool has multiple drives (it will stripe writes across disks if in a RAID-Z or stripe configuration).

Prefetch and Rust integration: Rust's standard file I/O (using `std::fs::File` and `read` calls) works fine on ZFS – it uses normal POSIX read system calls. There's no need for special asynchronous file I/O on ZFS unless you want to parallelize reads from the same file (rarely needed for a single sequential stream). If using `gen` for disk reading, each `read()` call will simply trigger ZFS to serve the next chunk, possibly from ARC (if prefetched) or from disk. If data was prefetched into ARC, the `read` will be satisfied from RAM at memory speed – very fast. If not, it will trigger a disk read and possibly additional prefetch. Thus, a **synchronous iterator reading from ZFS can still achieve asynchronous-like performance** because ZFS itself is doing async prefetch in the background. In many cases, you might find that a single-threaded reader can max out a fast disk array because ZFS is feeding it data from cache. This reduces the need for complicated multi-thread read-ahead logic at the application level.

In summary, Rust doesn't need any special treatment for ZFS beyond possibly tuning your ZFS dataset for large block sizes if you know your access is large-streaming. Long-lived sequential reads are a strong use-case for ZFS, and you can rely on it to **handle prefetching and caching aggressively** [16]. Just be mindful of memory: if you stream huge files that won't be reused, you might be filling ARC with data that won't be needed again. There are ZFS tunables (like `primarycache` setting metadata vs all data, or using `posix_fadvise` with `POSIX_FADV_DONTNEED` after reading) if that becomes an issue, but those are advanced tweaks. Generally, ZFS will manage ARC such that it doesn't jeopardize overall performance (e.g. it may not cache a giant file's every block in ARC if it detects one-pass streaming).

Composing an End-to-End Pipeline

With these tools, you can build a streaming pipeline that **pulls data from S3, caches it, and processes it in a streaming fashion** – all without Tokio-style async. One possible design is:

- **Stage 1:** A `may::go!` coroutine per input file that handles the S3 fetch. It opens an S3 HTTP stream (using a blocking client or socket wrapped for `may`), and reads, say, 8 MB at a time. For each chunk, it might immediately write it to the local ZFS file (so caching on disk happens as data arrives), and then send the chunk (or a handle to it) into a channel for processing. Writing to disk in the same coroutine is fine if the write is non-blocking or fast; if a write call blocks (say the ZFS write buffer is full), that coroutine yields – if processing is also in that coroutine, it would pause both download and processing, which isn't ideal. So you might instead **split download and disk-write into two coroutines** connected by a channel: one coroutine exclusively pulls from S3 and pushes data to an in-memory queue, another coroutine reads from that queue and performs the file write *and* further processing. This way, the S3 download isn't held up by local disk latency (except when the queue is full), and the processing isn't held up by network latency – they run in parallel.

- **Stage 2:** A processing stage that could be another coroutine (or even just the disk-writer coroutine itself doing double-duty). Once a chunk is written to disk (or as it's written), you can process it – e.g. parse records, transform data, etc. This processing could also be structured as an iterator. For example, if Stage 2 is single-threaded, you might simply iterate over the channel of chunks. You can convert a channel receiver into an iterator by repeatedly calling `recv` in the `Iterator::next` (blocking until a chunk is available). This is a common pattern to bridge asynchronous producers with pull-based consumers. The **Rust standard library MPSC** has a `.iter()` adapter that turns it into an iterator of messages, and libraries like Crossbeam have similar patterns. In the case of `may`, its channel can be used similarly (calling `.recv()` will park the coroutine if empty, which is what we want for backpressure). So you *can* still use iterator-style composition in the processing stage, even if the data arrival is driven by an async task. For instance, your Stage 2 coroutine could simply do: `for chunk in chunk_receiver { process(chunk); }`. This loop under the hood calls `recv` and yields when there are no chunks, seamlessly integrating with the `may` scheduler.

- **Stage 3:** If there's an output stage (perhaps writing results out or uploading to another service), you could again use either synchronous code (if it's minor or can buffer) or another coroutine if it involves I/O that could block. The pipeline can thus have multiple stages connected by channels, essentially forming a *streaming pipeline with backpressure*. This design is very much like Go's pipeline pattern (goroutines + channels), implemented in Rust via `may`. The advantage is clarity and the ability to use normal control flow in each stage, while the runtime handles scheduling.

Crucially, all these stages can be **composed or orchestrated fairly easily**. For example, you might spawn N S3-reading coroutines for N files in parallel. Each writes to its own file on ZFS and sends chunks to its own processing coroutine, or perhaps to a thread-safe work queue if you want a pool of processors. There's flexibility to design the workflow. The *simplest* might be one task per file that does everything: read from S3, write to disk, process, yield results – written as a straightforward sequential function but internally yielding at I/O points. This is essentially what one would do with async/await, but here `may` can achieve it in a single function using blocking calls that yield. On the other hand, you could also integrate the `gen` iterator approach for the disk read portion *after* caching: e.g. if you choose not to process streaming, you could just download to disk (with `may` for concurrency) and later use a `gen`-based iterator to stream from the cached file for processing. In that case, the pipeline is two-phase (download phase, then processing phase) rather than fully streaming. The question hints at a streaming approach, so likely we want to process as we download.

Composition with Iterator Chains: If you go the generator route for reading, you can indeed directly plug it into iterator combinators for processing. For instance, one could express a pipeline like:

```
let results: Vec<MyResult> = read_chunks(local_file)  // gen-based iterator
    .map(|chunk| transform(chunk))
    .filter(|item| item.is_valid())
    .collect();
```

This is a **pull-based, streaming pipeline** entirely in Rust's iterator paradigm. It's simple and has no explicit concurrency – which might be fine if I/O latency is low (local disk) and processing is not slow relative to I/O. It is easy to **reason about backpressure** here: `.collect()` will pull chunks one by one, so at most one chunk (or a small buffer of them inside combinators) is in flight. The Kevin Hoffman example demonstrates this style, reading 1MB chunks from a file and sending them via gRPC, partitioning results into success/failure lists using just iterator adapters [20] [21]. Generators make such code concise and clear, equivalent to writing a `for` loop but more composable.

Composition with `may`: In a `may` coroutine world, composition is a bit more manual because Rust doesn't (yet) have a built-in concept of *async iterators* in the sync context (the `Stream` trait exists for async, but we're avoiding async/await here). So you typically end up writing loops and sending messages. However, you can still create abstractions. For example, you could write a function that returns a channel receiver as a stream of chunks, and another function that consumes from a receiver and yields processed items. The absence of compiler-enforced async streams means you manage the life cycle of tasks and channels yourself. This isn't necessarily bad – it gives you control. Many developers structure such code with clearly named stages and use `may::join!` to wait for tasks or use a coordinating thread to spawn and monitor coroutines.

One should also note that `may` coroutines can interoperate with traditional threads if needed. For instance, you might dedicate one OS thread to run all `may` tasks (single-threaded scheduler for coroutines) and use another OS thread for heavy CPU processing (perhaps via Rayon's thread pool or just spawn threads for CPU-bound work). Communication between them could be via channels as well. This kind of hybrid approach might be useful if, say, parsing the data is CPU-intensive – you could feed chunks from the `may` pipeline into a Rayon parallel iterator for CPU processing, then collect results. Since the question focuses on avoiding Tokio and async-std, leveraging threads for parallel CPU is still on the table (Tokio isn't required for parallelism, only for async I/O). Rust's standard threads are quite efficient for moderate numbers, but if you needed tens of thousands of tasks, `may` would have the edge.

Finally, regarding **community insights and practical use**: It's worth mentioning that Rust's async ecosystem is far more common than stackful coroutines, so using `may` is a niche approach. Developers note that you have to "go all in" with it – you won't easily mix `may` coroutines with, say, a library that provides a `Future` or an `async fn` [22]. In our pipeline scenario, this is fine because we control the code and can choose appropriate libraries (or even implement the S3 HTTP GET with a simple HTTP client that can work in blocking mode). The advantage we gain is simplicity in our own code and possibly performance benefits by avoiding certain overheads. The disadvantage is fewer out-of-the-box libraries (for example, the official AWS SDK for Rust is async-focused and wouldn't directly work under `may` without spawning separate threads). Some community members have demonstrated that using synchronous generators (like `gen` blocks) yields very clean code for streaming tasks and could match the performance of async code [2]. In fact, once Rust stabilizes `gen`, it will allow writing code that *"looks synchronous as well [as `may`], but with better performance"* than stackful coroutines [2]. So the trend is that Rust is enabling more of this style natively.

Bottom Line: Both approaches can be effective for building a high-performance ELT pipeline:

- **Generators (`gen` blocks)** are great for implementing *readers that yield data chunks lazily*, integrating naturally with Rust iterator chains and ensuring inherent backpressure. They keep things synchronous and simple, which is often sufficient when dealing with a single stream from a fast local cache. The overhead is negligible – you basically pay only for the I/O syscalls and whatever processing you do, not for the abstraction itself [2].

- `may` **Coroutines** shine when you need *overlap* – e.g. downloading while processing, handling multiple streams concurrently – without introducing a full async runtime. They allow your code to remain in a familiar imperative style while the runtime handles scheduling around blocking points [8]. For read-heavy workloads where latency hiding is key (like waiting on an S3 response), this can improve throughput. Just be mindful of the library's requirements (stick to its I/O types or patterns) and the slightly higher per-task cost. In practice, `may` has shown it can handle massive concurrency with performance comparable to Tokio in I/O heavy workloads [11], so it's a viable alternative for specialized systems.

By leveraging ZFS's robust read-ahead and caching, plus Rust's efficient generators or coroutines, you can create a **streaming pipeline that is both fast and memory-efficient**. For example, you might implement a local cache reader as a generator that yields cached chunks (if the file is already on ZFS), and a fallback mechanism that uses `may` coroutines to fetch from S3 and write to disk when a cache miss occurs. This combination avoids tying your code to async "colored" functions and can be easier to reason about synchronously, while still achieving asynchronous throughput. It's a powerful approach, especially in an environment where reliability (Rust's safety, ZFS's data integrity) and performance (parallel I/O, no needless buffering) are top priorities.

References and Community Sources

- Rust RFC for `gen` blocks (Rust 2024 Edition) – introducing generators that produce iterable values [4] [1].
- Kevin Hoffman, *"Creating a Stream-Chunking Iterator in Rust"* – example of reading a file in 1MB chunks and processing via iterators [3] [20].
- Reddit discussion on **Rust stackful coroutines (`may`)** – highlights ease of writing sync-like code and trade-offs in performance vs async/await [8] [2].
- Rust Internals thread on stackful coroutines safety – notes `may` uses assembly for context switching and the TLS caveat [14].

- Documentation of an alternative coroutine library comparing performance – shows stackful coroutines (`may`) vs Tokio are in similar ballpark for I/O workloads [11].
- ZFS community discussions – explain how ZFS prefetch and record size affect streaming read performance (large record sizes and sequential access yield bigger, more efficient reads) [17] [16].

---

[1] [4] [5] [6] 3513-gen-blocks - The Rust RFC Book
https://rust-lang.github.io/rfcs/3513-gen-blocks.html

[2] [8] [9] [13] [15] [22] How does May, the stackful coroutine library, help in asychrounous IO? : r/rust
https://www.reddit.com/r/rust/comments/7obkys/how_does_may_the_stackful_coroutine_library_help/

[3] [20] [21] Creating a Stream-Chunking Iterator in Rust | by Kevin Hoffman | Medium
https://kevinhoffman.medium.com/creating-a-stream-chunking-iterator-in-rust-d4063ffd21ed

[7] [10] may - Rust
https://docs.rs/may/latest/may/

[11] GitHub - losfair/rust-coroutines: Stackful coroutine library for Rust that "just works"
https://github.com/losfair/rust-coroutines

[12] [14] What is the current safety story for library-based stackful coroutines? - Unsafe Code Guidelines - Rust Internals
https://internals.rust-lang.org/t/what-is-the-current-safety-story-for-library-based-stackful-coroutines/14863

[16] [17] [18] Does anyone know of detailed articles on how ZFS Prefetch works compared to other file system's prefetch? : r/zfs
https://www.reddit.com/r/zfs/comments/plvadq/does_anyone_know_of_detailed_articles_on_how_zfs/

[19] [SOLVED] - Slow ZFS performance - Proxmox Support Forum
https://forum.proxmox.com/threads/slow-zfs-performance.51717/