DESIGN of Gene Lang

1. Overview and Goals

1. **Language**: A static typed Gene language, interpreted at runtime.
2. **Implementation Language**: Zig, to leverage its low-level control, minimal runtime, and straightforward concurrency primitives.
3. **Core Modules**:
    - `ast.zig`: Abstract Syntax Tree definitions
    - `parser.zig`: Parsing `.gene` source
    - `typechecker.zig`: Static checks
    - `bytecode.zig`: Bytecode instructions for the interpreter
    - `vm.zig`: The interpreter (virtual machine)
    - `main.zig`: CLI or main entry
4. **Concurrency**:
    - **Multiple OS threads** (each with a separate VM)
    - **Green threads** (fibers) within each VM for cooperative concurrency

With this approach, we aim for:

- Simple scaling across multiple CPU cores (one VM per OS thread).
- Cooperative concurrency within each VM (lightweight green threads).

---

2. Project Layout

A typical folder structure:

```
static-gene-zig/
 ├─ build.zig
 ├─ src/
 │   ├─ ast.zig
 │   ├─ parser.zig
 │   ├─ typechecker.zig
 │   ├─ bytecode.zig
 │   ├─ vm.zig
 │   └─ main.zig
```

```
├─ tests/
│    ├─ parser_tests.zig
│    ├─ typechecker_tests.zig
│    ├─ vm_tests.zig
│    └─ ...
└─ README.md (this DESIGN.md can live here or be separate)
```

build.zig

- Defines how to compile the project.
- May create an executable `gene-compiler` from `src/main.zig`.
- Optional: add multiple `b.addTest(...)` steps for unit tests.

src/

1. **`ast.zig`**
   - Structures for `GeneAstNode`, `ClassDecl`, `FnDecl`, etc.
   - Minimal logic, just data definitions.
2. **`parser.zig`**
   - A function `parseGeneSource(allocator, source)` → returns `[]AstNode`.
   - Possibly uses a custom or library-based approach to parse.
3. **`typechecker.zig`**
   - `checkProgram(nodes: []const AstNode)` → performs static checks.
   - May produce a typed version or annotate nodes.
4. **`bytecode.zig`**
   - Defines `OpCode` enum/union, e.g. `LoadInt`, `AddInt`, etc.
   - `BytecodeFunction` struct with instructions array.
   - `lowerToBytecode(typedAstOrSimilar)` for generating specialized instructions.
5. **`vm.zig`**
   - `pub fn VM() type { ... }` constructing an interpreter that runs the bytecode.
   - `runFunction(func: BytecodeFunction)` iterates instructions.
   - Possibly includes **fiber logic** for green threads or references an external fiber library.
6. **`main.zig`**
   - The CLI entry point.
   - Typically: parse → typecheck → bytecode → run in VM.
   - Also spawns multiple OS threads if user wants parallel usage.

tests/

- Each file can hold `test "..." { ... }` blocks.
- `parser_tests.zig`, `vm_tests.zig`, etc.

---

3. Concurrency Model

3.1 Overall Approach

- **Multiple OS threads**: each owns its own `VM` instance.
    - Minimizes shared-state complexity. Each thread has a separate `ast.zig` parse or distinct tasks.
    - If needed, pass data among threads via message queues or a minimal channel.
- **Green threads** within each VM**:
    - Each OS thread can run multiple “user-level” tasks or coroutines *cooperatively*.
    - Achieved by either:
        1. Zig `async/await` approach if code is primarily I/O-bound.
        2. A custom fiber system (cooperative yield) that keeps multiple call stacks.
    - The interpreter is aware of multiple “logical threads” that share the same VM state but yield to each other to avoid blocking.

3.2 Implementation Outline

1. **OS Thread**: The function that creates a `VM` instance (`VM.init()`) and enters a loop for receiving tasks or script references to run.
2. **Green Threads**: Within that VM, the user code can spawn multiple tasks if the language offers concurrency constructs (`spawn`, `async fn`, etc.).
    - The VM fiber scheduler manages switching among those tasks.
    - One approach: define an opcode `OP_YIELD` or a builtin function `(yield)` that cooperatively yields to the VM scheduler.
3. **Synchronization**: If only the single VM is touched by these fibers, no locks are needed. They all share the same memory but cannot truly run in parallel—only one fiber runs at a time *per OS thread*.
4. **Parallel** across OS threads**: Each OS thread can fully run on a different CPU core. So if you have 4 OS threads, you get 4 actual parallel VMs. Within each, you have local concurrency with green threads.

---

4. Detailed Steps

4.1 Start-up

1. **Parse CLI** in `main.zig`: decide how many OS threads to spawn (`thread_count`).
2. **Compile** or **load** the `.gene` scripts. Possibly parse them once, store separate bytecode modules for each script.

4.2 OS Thread Creation

```
fn threadEntry(threadIndex: usize, module: BytecodeModule) !void {
    // create VM
    var my_vm = vm.VM.init();

    // run or schedule tasks
    // e.g. my_vm.runFunction(module.functions[0])
    // plus handle green-thread tasks
}
```

In `main.zig`, something like:

```
for (0..thread_count) |i| {
    threads[i] = try std.Thread.spawn(threadEntry, i);
}
// join them
```

4.3 Green Threads Scheduling

Inside `vm.zig`, you store an optional **fibers** array or a small queue:

```
pub const VM = struct {
    // e.g. array of fibers
    pub fn init() VM {
        return .{};
    }

    pub fn runFiber(...) !void {
        // pick fiber, run instructions until yield or done
    }
```

```
    pub fn scheduleFibers() !void {
        // e.g. while !all_done: run each fiber in round-robin
    }
};
```

If using Zig's `async/await`, you can do:

```
// single-threaded async
pub fn runTasks() !void {
    var t1 = async doTask1();
    var t2 = async doTask2();
    // ...
    await t1;
    await t2;
}
```

But for a user-level "Gene concurrency" model, you might write or wrap these in your interpreter instructions.

---

5. Memory Model and Safety

1. **No Shared Data** across VMs: Easiest approach, each VM is allocated with local structures.
2. **Green Threads** share the same local environment:
    - They rely on cooperative yields, so the interpreter only runs one fiber at a time.
    - No atomic overhead if you only have one OS thread per VM.
    - If your language has reference-counted or GC objects, ensure no data races since only one fiber runs at a time.

---

6. Potential Future Extensions

1. **Message Passing** among VMs:
    - If you want cross-VM communication, define channels or a bridging object.
    - Possibly a built-in function `(send msg to VM #2)`, `(receive)`. This can be done with `std.Thread.Channel` or a custom queue.
2. **JIT or Partial AOT**:

    - Possibly integrate with a library like LLVM (via C FFI) or Cranelift.
    - Doesn't change concurrency model but could speed up hot code.
3. **GPU Offloading**:
    - Provide built-in `(gpu_call data...)` that calls an external CUDA/OpenCL/Vulkan function in your runtime.
4. **Macros** or Compile-Time** expansions can be done if you have a compilation pipeline that transforms AST. Not strictly concurrency-related, but part of language design.

---

7. Example Implementation Order

1. **Implement** single-thread single-fiber version (no concurrency).
    - `parser` → `typechecker` → `bytecode` → `vm.runFunction(...)`.
    - Confirm correctness on a single script.
2. **Add** multi-VM approach:
    - Spin up multiple OS threads, each with `VM.init()`.
    - Possibly feed each VM a separate script or job.
3. **Add** green threads** in the VM:
    - Either use a Zig fiber library or create a small manual fiber system.
    - Let the user code `(spawn ...)` or `async` to create multiple tasks that run in that single VM.
4. **Test** concurrency with integration tests.
    - E.g., produce scripts that create tasks, do yields, etc., and see if results are correct.

---

8. Summary

- **Project Layout**: `ast.zig`, `parser.zig`, `typechecker.zig`, `bytecode.zig`, `vm.zig`, `main.zig`.
- **Concurrency**:
    - **N** OS Threads, each pinned to its own VM (no shared mutable data).
    - Inside each VM, **K** green threads for concurrency. They share the same interpreter data but only one runs at a time on that OS thread → no data races.
- **Implementation** in Zig:
    - Use `std.Thread` to spawn OS threads.
    - Use either Zig's `async` or a custom fiber library for green threads.
- **Result**: A **static interpreted** language that can scale across cores (multiple OS threads) and handle concurrency inside each VM with green threads.

This design ensures:

1. **Parallelism** across OS threads on multi-core CPUs.
2. **Lightweight concurrency** at the user level via green threads.
3. Minimal locking overhead, as each VM is effectively isolated to one OS thread.

Hence, this approach is strongly recommended if your language scenario can be partitioned into tasks or scripts that don’t heavily share data among VMs.