Refactoring sourcekitd to Use Swift Concurrency

Introduction and Background

Problem

sourcekitd is a critical component of the Swift toolchain, enabling features like code completion, syntax highlighting, and diagnostics by interfacing with the compiler internals. However, **sourcekitd** currently relies on **C++ for concurrency**, which can be:

Fragile: Lock-based concurrency introduces race conditions if not carefully managed.
Difficult to Maintain: C++ thread handling and global locks spread across multiple files can lead to complex logic.
Platform Dependent: On macOS, sourcekitd is run via XPC (separate process), but on other platforms it remains in-process, making crash resilience harder.

Why Refactor?

With Swift 6 concurrency (actors, async/await, structured concurrency), we now have language-level tools to manage concurrency safel.. Refactoring **sourcekitd** to Swift concurrency will therefore:

Improve Race Safety: Actors isolate state, eliminating the need for manual locking.
Enhance Maintainability: Concurrency code becomes explicit and more readable, reducing the risk of subtle bugs.
Enable Out-of-Process Models on All Platforms (stretch goal): A robust concurrency abstraction could ease separating sourcekitd into a dedicated process on Linux and Windows, mirroring macOS's XPC model and improving crash resilience in SourceKit-LSP.

Current State

`Requests.cpp`: Contains logic for handling requests such as code completion, syntactic/semantic queries, indexing, etc. Uses `std::thread`, `std::mutex`, and callback-based concurrency.
`SwiftASTManager.cpp`: Manages AST caching and parsing. It often involves a global or class-level mutex to coordinate parallel requests.
Global Locks and Thread Pools: Present across the codebase for concurrency control.

Project Goals

Migrate Request Handling from C++ Threads/Callbacks to Swift async/await

1. Develop Swift APIs that replace or wrap the existing thread-based dispatch model in `Requests.cpp`.
2. Ensure the new approach remains functionally compatible (no regressions!).

Introduce Actor-Based AST Management

3. Replace the **lock-based** model in `SwiftASTManager.cpp` with a **Swift actor**, isolating shared AST caches from data races.
4. Wrap existing C++ parsing logic where needed, but **remove** direct concurrency constructs (e.g., `std::mutex`).

Maintain or Improve Performance

5. Validate through existing test suites and possible new stress/concurrency tests.
6. Ensure no major regressions in code-completion latency adn memory usage.

Scope Clarification:

- **Included**: Converting major concurrency hotspots (request dispatch, AST caching) to Swift concurrency.
- **Not Included?**: A complete rewrite of all performance-critical C++ logic to Swift (we will rely on bridging for some parts).

Implementation

Below, we detail the technical plan to achieve a safe concurrency model in Swift. We also show **code snippet examples** from `Requests.cpp` and `SwiftASTManager.cpp` that illustrate the **before** (C++ concurrency) and **after** (Swift concurrency) transformations. These aren't tested, I have only been playing around with these!

Swift Concurrency Wrappers for Requests (Requests.cpp)

Current State

```
bool handleRequest(const Request &Req, Response &Resp) {

  switch (Req.Kind) {

    case RequestKind::Completion:
```

```
    // Spawns thread or uses global locks
    std::thread([&]() {
      auto results = doCodeCompletion(Req.filePath);
      Resp.completionResults = results;
    }).detach();
    return true;

  // ... other request kinds ...
 }
 return false;
}
```

Refactored

```
actor RequestActor {
    func handleRequest(_ req: Request) async -> Response {
        switch req.kind {
        case .completion:
            // Use async bridging to C++ function
            let results = await doCodeCompletionInCPP(req.filePath)
            return Response(completionResults: results)
        // ... other kinds of requets
        }
    }

    // "bridging" function
```

```
    private func doCodeCompletionInCPP(_ filePath: String) async -> [String] {
        return await withCheckedContinuation { continuation in
            cppDoCodeCompletion(filePath) { completionList in
                continuation.resume(returning: completionList)
            }
        }
    }
}
```

Key points:

- Remove **manual threads** (`std::thread`/`detach`) and use **Swift async** tasks.
- **`withCheckedContinuation`** safely bridges the callback from C++ into Swift concurrency.

Actor-Based AST Manager (SwiftASTManager.cpp)

Current State

```
std::mutex ASTMutex;

std::unordered_map<std::string, ASTNode*> ASTCache;

ASTNode* SwiftASTManager::parseAST(const std::string &filePath) {
  std::lock_guard<std::mutex> lock(ASTMutex);
  auto it = ASTCache.find(filePath);
  if (it != ASTCache.end()) {
    return it->second;
  }
  ASTNode* newAST = doParse(filePath);
  ASTCache[filePath] = newAST;
```

```
  return newAST;
}
```

Refactored

```
actor SwiftASTManager {
    private var astCache: [String: ASTNodeRef] = [:]

    func parseAST(_ filePath: String) async -> ASTNodeRef {
        if let existing = astCache[filePath] {
            return existing
        }
        let newAST = await withCheckedContinuation { continuation in
            cppDoParse(filePath) { astPtr in
                continuation.resume(returning: ASTNodeRef(astPtr))
            }
        }
        astCache[filePath] = newAST
        return newAST
    }

    func clearCache() {
        astCache.removeAll()
    }
}
```

Main points:

- **No explicit locks**—the Swift actor ensures only one task accesses `astCache` at a time.
- **ASTNode** may remain partially in C++ if rewriting the parser fully in Swift isn't feasible.

Optional: Cross-Platform Out-of-Process Execution

- Evaluate possibility of **separate processes** on Linux/Windows akin to macOS XPC.
- Swift concurrency remains the same internally, but requests would cross process boundaries.
- Document or prototype if time permits (stretch goal).

Timeline

Phase 1 (Weeks 1–2): Setup & Swift Concurrency Wrappers

Introduce Swift "bridging" layer for `Requests.cpp`:

1. Wrap major request types (e.g., code completion) in `async` Swift functions.
2. Confirm existing tests pass.

Deliverable: Swift "RequestActor" that replaces manual threads in a minimal bridging approach.

Phase 2 (Weeks 3–5): Actor-Based AST Management

Refactor lock-based AST code (`SwiftASTManager.cpp`) into a Swift actor.

1. Encapsulate AST data in an actor, remove `std::mutex`.

Deliverable: A functional `SwiftASTManager` actor with robust concurrency tests.

Phase 3 (Weeks 6–8): Testing, Stabilization & Optional Out-of-Process

Testing & Performance:

1. Run concurrency stress tests and performance benchmarks to detect regressions.
2. Fix bugs, refine bridging code.

Optional: Investigate **out-of-process** models for Linux/Windows.

3. Document or prototype feasible approaches if time permits.

Deliverable:

- FInal code, fully tested.
- Documentation on how Swift concurrency is now used in sourcekitd.
- Optional PoC for cross-process operation.

Summary

Problem: sourcekitd's C++ concurrency model is complex and error-prone.
Solution: Migrate to Swift concurrency (actors, async/await), as shown in the **code examples** from `Requests.cpp` and `SwiftASTManager.cpp`.
Benefits: Increased maintainability, reduced data races

Why Me

After working on the kernel code at VMware for 2 years in the network datapath team, I have been fascinated with low latency networking and the Swift on Server and the Swift NIO project have been very interesting to me. I have been able to contribute to it [1][2], and in the process learnt a ton about Swift's internals, and even used Swift Concurrency in interesting ways to develop features like async packet captures [3].

I want to continue contributing to the project, and this particular project aligns really well with my goals around learning Swift Concurrency primitives deeply, and further, make Swift a general purpose language for building high performance servers.

I have also had the opportunity to give some talks around interesting concurrency constructs like Coroutines [4], and will soon be giving a talk in GopherCon NA around Race Detection, and writing code that is Sequentially Consistent, so this is an area that is very close to my interests.

Currently, I'm a Masters student at UCL where I'm working on Formal Verification, particularly applied to complex systems like Distributed Systems and Compilers.

References

[1] Peek API Variants: https://github.com/apple/swift-nio/pull/3160

[2] Peek Integer: https://github.com/apple/swift-nio/pull/3157

[3] Async PCAP: https://github.com/apple/swift-nio-extras/pull/253

[4] Coroutines Talk: https://www.youtube.com/watch?v=sDTJMm__DXE

Questions For Mentors

Q 1: To understand the motivation better, which areas of sourcekitd are most prone to concurrency issues or race conditions right now? Are there particular locks or data structures in `Requests.cpp` and `SwiftASTManager.cpp` that you know are causing contention or complexity?

Q 2: How is the lifetime of the AST data currently managed? Are there reference-counting or memory-management patterns we need to preserve when moving to Swift concurrency? Do we ever share AST objects between multiple threads and do we need a new approach to safely share them in Swift concurrency?

Q 3: Are there any submodules or functionalities that must remain in C++ for compatibility reasons, and can't be fully converted to Swift concurrency? How 'deeply' should we restructure code in `Requests.cpp` and `SwiftASTManager.cpp` - do you foresee only concurrency changes or also broader re-architecture?

Q 4: Some clarifying questions:

a) Will we have to change the representation of AST node pointers when bridging to Swift, or is a simple 'pointer-wrapping' approach acceptable?

b) Does the existing C++ library code we call from Swift concurrency assume it's called from a single thread, or can it handle concurrency from multiple Swift tasks?

c) Do you envision one big actor for AST management or multiple fine-grained actors (e.g., separate actors for caches, indexing, completions, etc.)?