Interview Review Chart

Karl Solomon

December 30, 2024

Contents

Part I
Distributed Systems

1 Terms

CAP Consistency, Availability, and Partition Tolerance. Consistency and Availability are tradeoffs. You cannot have perfect/complete of both. System design should depend on how much of each to prioritize.

ACID Atomicity, Consistency, Isolation, and Durability

Consistency If you have multiple copies of data they should be the same across locations.

Eventual Consistency Eventual consistency is a consistency model used in distributed systems to ensure that, given enough time without new updates, all replicas of a data item will converge to the same value. It allows for temporary inconsistencies but guarantees that all nodes will eventually become consistent. This model is often used in systems where high availability and partition tolerance are prioritized over immediate consistency.

2 Reasons for having a DS

- Single server means single point of failure. A single outage/attack, failure leads to failure of whole system.
- Unable to scale up compute to match lots of users. Cost of Vertical Scaling is high/limiting.
- Latency is high for (geographically) distant users.

3 Optimizing for Consistency

3.1 Two Generals Problem

The Two Generals Problem is a thought experiment that demonstrates the difficulty of achieving consensus between two parties over an unreliable communication channel. It involves two generals who must coordinate an attack but can only communicate via messengers who may be intercepted. The problem highlights the impossibility of guaranteeing message delivery and agreement in such a scenario, illustrating the challenges of achieving reliable communication in distributed systems.

3.2 2-phase commit

1. Leader sends "prepare"
2. Follower sends "ack"
3. If leader fails to receive ack for any follower, then it must abort/rollback the change. Each follower has a timeout after prepare upon which they will execute the rollback if they never receive "commit" message.
4. Else (all acks received), then leader "commits" internally
5. Leader sends "commit"
6. Follower sends "ack"

3.3 Eventual Consistency

4 Locking

4.1 Eventual Locking

- Redlock Algorithm (Redis):

Part II
Parallelism

5 Terms

Concurrency When two or more tasks can start/run/complete in overlapping time periods. This does not necessarily mean they'll be running in overlapping time periods. Examples include:
RTOS

Parallelism When tasks run at the same time (e.g. on a multi-core CPU)

Multithreading When multiple tasks are running on a CPU. This can be implemented truly parallel where each task has access to separate HW/core. However, more common in desktop applications is SMT.

SMT (Simultaneous Multithreading) Multiple threads share 1 core. The thread instructions are pipelines s.t. they run mostly in-parallel, and when one is waiting for I/O the other can run uninhibited, however since only one thread can access a dedicated HW block at any given time they are not truly parallel.

6 Common Concurrency Problems

Atomicity-Violation The desired series of multiple memory accesses is violated (and not enforced).

```
1  T1:
2  if(obj->info)
3    print(obj->info)
4
5  T2:
6  obj->info = NULL
```

Order-Violation The desired order of two memory accesses is reversed, and not enforced.

```
1  T1:
2  void init()
3    mThread = createThread()
4
5  T2:
6  var = mThread->info // it is possible this runs before
       createThread
```

Deadlock Requires: Mutual Exclusion, No Preemption, Hold-and-wait, and circular wait

7 Concurrency and Multithreading

7.1 Basic concepts of threads and processes

7.2 Thread synchronization mechanisms (mutexes, semaphores, locks)

7.3 Race conditions:

1. Multiple threads accessing a shared resource
2. At least one thread writes to the resource
3. Lack of proper synchronization

7.4 deadlocks

```
std::mutex m1, m2;
// Must have 2 locks where they are locked in different orders in different locations/threads
thread1: m1.lock(); m2.lock(); m1.unlock(); m2.unlock();
thread2: m2.lock(); m1.lock(); m2.unlock(); m1.unlock();
```

7.5 Atomic operations

```
std::atomic<T> var; // where T is a primitive type
var = val;
var.load(val);
var.store(val);
var.wait(); // waits until the value changes
T current = var.exchange(val); // writes val to var and gets previous/current value of var
// Compare-and-swap. Atomically compares object.value with that of expected. If bitwise-equal then replaces former with desired. Otherwise loads actual value into expected (via load operation)
bool res = var.compare_exchange_strong(expected, desired); // preferred when don't expect high contention and cost of retrying is significant. Simpler, but slower.
bool res = var.compare_exchange_weak(expected, desired);  // preferred if you're anyways retrying in a loop or cost of retrying is low. More efficient, but more complex.
```

```
// Weak usage
std::atomic<int> value{0};
int expected = 0;
int desired = 1;
while(!value.compare_exchange_weak(expected, desired)){
  ; // handle spurrious failures
}

// Strong usage
if(value.compare_exchange_strong(expected, desired)){
  // op successful
}
else
{
  // op failed
}
```

8 Thread-Safe Data Structures:

8.1 Concurrent collections

8.2 Concurrent collections (e.g., ConcurrentQueue, ConcurrentBag)

8.3 Lock-free data structures

8.4 Understanding the differences between thread-safe and non-thread-safe collections

9 Design Patterns for Concurrency:

9.1 Producer-Consumer pattern

9.2 Readers-Writer pattern

9.3 Thread pool pattern

10 Language-Specific Concurrency Features for C++:

10.1 std::thread

10.2 <atomic>

11 Callback Mechanisms:

11.1 Function pointers

```c
#include <stdio.h>

// Define a struct with function pointers for arithmetic operations
typedef struct {
    int (*add)(int, int);
} ArithmeticOperations;

// Define the functions for arithmetic operations
int add(int a, int b) { return a + b; }

int main() {
    // Initialize the struct with function pointers
    ArithmeticOperations ops;
    ops.add = add;
    // Use the function pointers to perform operations
    int x = 10, y = 5;
    printf("Add: %d + %d = %d\n", x, y, ops.add(x, y));
    return 0;
}
```

11.2 Delegates (in languages that support them)

11.3 Lambda expressions

12 Performance Considerations:

12.1 Understanding the overhead of different synchronization mechanisms

Generally best practice to measure performance in single-threaded vs multithreaded/parallel environments. Since there is overhead with creating/cleaning up threads/processes it can make your program run slower in smaller data sets.

12.2 Balancing thread safety with performance

13 Testing Multithreaded Code:

13.1 Techniques for writing unit tests for concurrent code

13.2 Tools for detecting race conditions and deadlocks

- **Helgrind**: Part of Valgrind suite. Checks for race conditions, but slow.
- **ThreadSanitizer**: Compiler flag in llvm/clang. Faster than Helgrind.
- **RacerD**: Meta's C++-specific concurrent static analyzer. Good for larget code-bases.
- **Clang Static Analyzer**: Detects some simple conditions.

14 Algorithms for Concurrent Operations:

14.1 Compare-And-Swap (CAS) operations

14.2 Lock-free algorithms

15 Memory Models:

15.1 Understanding memory barriers and volatile variables

Barrier/Fence Ensure that memory operations are not reordered across the fence by the compiler (which could lead to unexpected behavior). This is necessary b/c modern CPUS/compilers reorder instructions for performance optimization.

std::atomic_thread_fence(std::memory_order::<order>; is the most general barrier. cppref. List of memory orders:

- memory_order_relaxed: No ordering guarantees. Guarantees atomicity.
- memory_order_acquire: Prevents reordering of loads after the barrier. Useful for Consumers.
- memory_order_release: Prevents reordering of stores before the barrier. Useful for Producers.
- memory_order_acq_rel: Acquire + Release semantics. Useful for Read-modify-write operations.
- memory_order_seq_cst: Strongest ordering. Ensures sequential consistency across all threads.

```
std::atomic<int> data(0);
std::atomic<bool> ready(false);

// Memory order: memory_order_relaxed
// Use case: No synchronization or ordering constraints, just atomicity.
// Example: Incrementing a counter where order doesn't matter.
std::atomic<int> counter(0);
std::thread t1([&]() {
    for (int i = 0; i < 1000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
});
std::thread t2([&]() {
```

```
    for (int i = 0; i < 1000; ++i) {
        counter.fetch_add(1, std::memory_order_relaxed);
    }
});
t1.join();
t2.join();
// Result: counter = 2000 (order of increments doesn't matter)

// Memory order: memory_order_consume
// Use case: Data dependency ordering, rarely used due to complexity.
// Example: Reading a value that depends on another atomic load.
std::atomic<int*> ptr(nullptr);
int value = 42;
std::thread t3([&]() {
    ptr.store(&value, std::memory_order_release);
    ready.store(true, std::memory_order_release);
});
std::thread t4([&]() {
    while (!ready.load(std::memory_order_consume));
    int* p = ptr.load(std::memory_order_consume);
    if (p) {
        // Use *p
    }
});
t3.join();
t4.join();

// Memory order: memory_order_acquire
// Use case: Ensures that subsequent reads/writes are not moved
   before the load.
// Example: Reading a flag to ensure data is ready.
std::thread t5([&]() {
    while (!ready.load(std::memory_order_acquire));
    int d = data.load(std::memory_order_acquire);
    // Use d
});

// Memory order: memory_order_release
// Use case: Ensures that all previous writes are visible before the
   store.
// Example: Writing data before setting a flag.
std::thread t6([&]() {
    data.store(42, std::memory_order_release);
    ready.store(true, std::memory_order_release);
});
t5.join();
t6.join();

// Memory order: memory_order_acq_rel
// Use case: Combines acquire and release semantics.
// Example: Read-modify-write operations where both ordering
   constraints are needed.
std::atomic<int> shared_data(0);
std::thread t7([&]() { shared_data.fetch_add(1, std::
   memory_order_acq_rel); });
std::thread t8([&]() { shared_data.fetch_add(1, std::
   memory_order_acq_rel); });
t7.join();
t8.join();
// Result: shared_data = 2 (ensures correct ordering of operations)

// Memory order: memory_order_seq_cst
// Use case: Provides a single total order of operations, the
```

```
    strongest guarantee.
// Example: When strict ordering is required across threads.
std::atomic<int> seq_data(0);
std::thread t9([&]() { seq_data.store(1, std::memory_order_seq_cst);
    });
std::thread t10([&]() {
    int x = seq_data.load(std::memory_order_seq_cst);
    // Use x
});
t9.join();
t10.join();
```

15.2 Cache coherence issues in multi-core systems TODO (ksolomon): add info about cache banks, parity, and typical cache hit

/miss times.

Cache Coherence The process of ensuring that data is stored in multiple caches within a multiprocessor system is consistent and synchronized.

This ensures that all processors have a *consistent* view of shared memory. Cache coherence protocols manage the flow of data between caches, updating cache lines and tracking the status of shared data. This can be complicated because it requires balancing perforamnce and coherence overhead. The 2 main types of protocols are Directory-based and Snoop-based.

Directory-based The sharing status of a block of physical memory is kept in just one location (the directory). The direcotry can also be distributed to improve scalability. Communication is established using point-to-point requests through the interconnection network.

Snoop-based Every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, but no centralized state is kept. Caches are all accessible via some broadcast medium (a bus or switch), and all cache controllers monitor or *snoop* on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access. Requires broadcast, csince caching information is at processors. This is useful for small-scale machines.

Point of Coherency (PoC) Point at which all agents in a system which can access memory are guaranteed to see the same data.

Migration Data is migrated to the local cache levels.

Replication The same data is replicated across all caches.

Assume Snoop-based protocol. There are 2 ways to maintaint coherence:

1. **Write Invalidate Protocol:** Ensure that a processor has exclusive access to a data item before it writes that item. This is most common protocol.
2. **Write Broadcast/Update:** All cached copies are updated simultaneously. This requires more bandwidth. When multiple updates happen to the same location, unnecessary updates are done. However, this is a lower latency between write/read.

16 Practice Problems:

- Implement a thread-safe singleton

```cpp
#include <iostream>
#include <mutex>

class Singleton {
 public:
    // Delete copy constructor and assignment operator to prevent creation of additional instances through copying
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    // Static method to get the instance of the singleton
    static Singleton& getInstance() {
        static Singleton instance;  // Static Local variables are guaranteed to be thread-safe in C++11 and later
        return instance;
    }

    void showMessage() { std::cout << "Singleton instance accessed!" << std::endl; }

 private:
    // Private constructor to prevent instantiation
    Singleton() { std::cout << "Singleton instance created!" << std::endl; }
};

int main() {
    // Access the singleton instance
    Singleton& singleton = Singleton::getInstance();
    singleton.showMessage();

    return 0;
}
```

- Create a simple producer-consumer queue

```cpp
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <queue>
#include <thread>

class ThreadSafeQueue {
 public:
    void enqueue(int item) {
        std::lock_guard<std::mutex> lock(mutex_);
        queue_.push(item);
        cond_var_.notify_one();
    }

    int dequeue() {
        std::unique_lock<std::mutex> lock(mutex_);
        cond_var_.wait(lock, [this] { return !queue_.empty(); });
        int item = queue_.front();
        queue_.pop();
        return item;
    }

 private:
    std::queue<int> queue_;
    std::mutex mutex_;
    std::condition_variable cond_var_;
```

```
};

void producer(ThreadSafeQueue& queue, int numItems) {
    for (int i = 0; i < numItems; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));  // Simulate work
        queue.enqueue(i);
        std::cout << "Produced: " << i << std::endl;
    }
}

void consumer(ThreadSafeQueue& queue, int numItems) {
    for (int i = 0; i < numItems; ++i) {
        int item = queue.dequeue();
        std::cout << "Consumed: " << item << std::endl;
    }
}

int main() {
    ThreadSafeQueue queue;
    const int numItems = 10;

    std::thread producerThread(producer, std::ref(queue), numItems);
    std::thread consumerThread(consumer, std::ref(queue), numItems);

    producerThread.join();
    consumerThread.join();

    return 0;
}
```

- Implement a basic thread pool

```
#include <execution>  // Required for std::execution::seq
#include <iostream>
#include <numeric>
#include <random>
#include <thread>
#include <vector>

void accumulateRandomNumbers(int threadID) {
    static std::mutex m;
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(0.0, 1.0);

    std::vector<double> numbers(1024 * 1024);
    for (auto& num : numbers) {
        num = dis(gen);
    }

    double sum = std::reduce(std::execution::seq, numbers.begin(), numbers.end());
    std::lock_guard<std::mutex> lock(m);
    std::cout << "Thread " << threadID << " accumulated sum: " << sum << std::endl;
}

int main() {
    const int numThreads = std::thread::hardware_concurrency();
        // Number of threads in the pool
```

```
    std::vector<std::thread> threadPool;

    // Create and launch threads
    for (int i = 0; i < numThreads; ++i) {
        threadPool.emplace_back(accumulateRandomNumbers, i);
    }

    // Join threads to the main thread
    for (auto& thread : threadPool) {
        thread.join();
    }

    return 0;
}
```

- Dining Philosophers

```
#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

class DiningPhilosophers {
 public:
    DiningPhilosophers(int numPhilosophers)
        : numPhilosophers_(numPhilosophers), states_(
            numPhilosophers, State::THINKING) {}

    void philosopher(int id) {
        while (true) {
            think(id);
            pickUpForks(id);
            eat(id);
            putDownForks(id);
        }
    }

 private:
    enum class State { THINKING, HUNGRY, EATING };

    void think(int id) {
        std::cout << "Philosopher " << id << " is thinking." <<
            std::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds
            (1000));
    }

    void eat(int id) {
        std::cout << "Philosopher " << id << " is eating." << std
            ::endl;
        std::this_thread::sleep_for(std::chrono::milliseconds
            (1000));
    }

    void pickUpForks(int id) {
        std::unique_lock<std::mutex> lock(mutex_);
        states_[id] = State::HUNGRY;
        cond_var_.wait(lock, [this, id] { return canEat(id); });
        states_[id] = State::EATING;
    }

```

```
42      void putDownForks(int id) {
43          std::unique_lock<std::mutex> lock(mutex_);
44          states_[id] = State::THINKING;
45          cond_var_.notify_all();
46      }
47
48      bool canEat(int id) {
49          int left = (id + numPhilosophers_ - 1) % numPhilosophers_;
50          int right = (id + 1) % numPhilosophers_;
51          return states_[id] == State::HUNGRY && states_[left] !=
                State::EATING && states_[right] != State::EATING;
52      }
53
54      int numPhilosophers_;
55      std::vector<State> states_;
56      std::mutex mutex_;
57      std::condition_variable cond_var_;
58  };
59
60  int main() {
61      const int numPhilosophers = 5;
62      DiningPhilosophers diningPhilosophers(numPhilosophers);
63      std::vector<std::thread> threads;
64
65      for (int i = 0; i < numPhilosophers; ++i) {
66          threads.emplace_back(&DiningPhilosophers::philosopher, &
                diningPhilosophers, i);
67      }
68
69      for (auto& thread : threads) {
70          thread.join();
71      }
72
73      return 0;
74  }
```

Part III
Operating Systems

17 Virtualization

17.1 Scheduling

17.2 Memory

18 Concurrency

18.1 Mutex

18.2 Bugs

18.3 Events

19 Persistence

19.1 HDD/RAID

19.2 File Systems

19.3 Integrity

19.4 Distributed Systems

20 Security

21 Caches

TLB The TLB is a small, fast, and fast-access memory that is used to translate virtual memory addresses into physical memory addresses. a.k.a. Translation Lookaside Buffer.

22 Basic Concepts

- **TLB: Translate Lookaside Buffer**
- **Processes and Threads**
  - Process creation and termination
  - Thread lifecycle and management
- **Memory Management**
  - Virtual memory
  - Paging and segmentation
- **File Systems**
  - File system structure
  - File operations and permissions