Interview Review Chart

Karl Solomon

December 24, 2024

1 Parallelism

1.1 Concepts

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.

1.2 Implementation

- Concurrency and Multithreading
  - Basic concepts of threads and processes
  - Thread synchronization mechanisms (mutexes, semaphores, locks)
  - Race conditions:
    1. Multiple threads accessing a shared resource
    2. At least one thread writes to the resource
    3. Lack of proper synchronization
  - 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();
```

  - 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
```

```cpp
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.
```

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

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

- Thread-Safe Data Structures:
  - Concurrent collections
  - Concurrent collections (e.g., ConcurrentQueue, ConcurrentBag)
  - Lock-free data structures
  - Understanding the differences between thread-safe and non-thread-safe collections
- Design Patterns for Concurrency:
  - Producer-Consumer pattern
  - Readers-Writer pattern
  - Thread pool pattern
- Language-Specific Concurrency Features for C++:
  - std::thread
  - <atomic>
- Callback Mechanisms:
  - Function pointers
  - Delegates (in languages that support them)
  - Lambda expressions
- Performance Considerations:
  - 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.

  - Balancing thread safety with performance

- Testing Multithreaded Code:
  - Techniques for writing unit tests for concurrent code
  - 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.

- Distributed Systems Concepts:
  While not directly related to this problem, understanding concepts like eventual consistency and distributed locking can be beneficial

- Algorithms for Concurrent Operations:
  - Compare-And-Swap (CAS) operations
  - Lock-free algorithms

- Memory Models:
  - Understanding memory barriers and volatile variables
  - Cache coherence issues in multi-core systems

  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.

- 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

```cpp
#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;
}
```

- Solve classic concurrency problems like the dining philosophers problem

```cpp
#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;
    }

    void putDownForks(int id) {
        std::unique_lock<std::mutex> lock(mutex_);
        states_[id] = State::THINKING;
        cond_var_.notify_all();
    }

    bool canEat(int id) {
        int left = (id + numPhilosophers_ - 1) %
            numPhilosophers_;
        int right = (id + 1) % numPhilosophers_;
        return states_[id] == State::HUNGRY && states_[left]
            != State::EATING && states_[right] != State::EATING
            ;
    }

    int numPhilosophers_;
    std::vector<State> states_;
    std::mutex mutex_;
    std::condition_variable cond_var_;
};

int main() {
    const int numPhilosophers = 5;
    DiningPhilosophers diningPhilosophers(numPhilosophers);
    std::vector<std::thread> threads;

    for (int i = 0; i < numPhilosophers; ++i) {
        threads.emplace_back(&DiningPhilosophers::philosopher,
             &diningPhilosophers, i);
    }

    for (auto& thread : threads) {
        thread.join();
    }

    return 0;
}
```

Write a few test cases in addition to the solution Remember, for interviews, it's not just about knowing the solutions, but also being able to explain your reasoning, discuss trade-offs, and analyze the performance and correctness of your solutions. Lastly, be prepared to write code on a whiteboard or in a simple text editor. Practice implementing these concepts without relying on an IDE's features.

2 Operating Systems

2.1 Basic Concepts

- **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

3 C

3.1 Preprocessor

```
# // stringizes the macro parameter
#define stringify(x) #x
#define foo 1
stringify(foo) // --> evaluates to "foo", NOT "1"
```

```
## // concatenates the macro parameter
#define COMMAND(NAME) {#NAME, NAME ## _command}
struct command commands[] = {
  COMMAND(quit), // equivalent to {quit_command}
  COMMAND(help), // equivalent to {help_command}
}
```

- predefined macros
  - __FILE__
  - __LINE__
  - __DATE__
  - __TIME__
  - __STDC_VERSION__
  - __cplusplus
- item2
- item3
- item4

3.2 Peripherals

- **I2C**
  SDA is data, SCL is clock. PURs typically in the 1-4.7k range. Too weak = slow comm and errors. Clocks are usually 100k-1MHz. Addr can be 7 or 10 bit. This is rate-limiter for number of slaves, though line impedance would increase for each slave. Here are some use usage examples:

  1. Master sends START and slave Addr
  2. Master sends data to slave
  3. Master terminates with a STOP

  1. Master sends START and slave Addr
  2. Master sends data to slave
  3. Master sends repeatedSTART and either sends more data to slave or receives data from slave.

4. Master sends STOP

- **SPI**
  Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used for short-distance communication, primarily in embedded systems. It uses a master-slave architecture with a single master and multiple slaves. Communication is full-duplex, and it requires four wires: MOSI, MISO, SCLK, and SS.
- **UART**
  Universal Asynchronous Receiver-Transmitter (UART) is a hardware communication protocol that uses asynchronous serial communication with configurable speed. It is commonly used for communication between microcontrollers and peripherals. UART requires only two wires: TX (transmit) and RX (receive).
- **USB**
  Universal Serial Bus (USB) is an industry-standard for short-distance digital data communications. It supports plug-and-play installation and hot swapping. USB is used for connecting peripherals such as keyboards, mice, printers, and external storage devices to computers.
- **HDMI**
  High-Definition Multimedia Interface (HDMI) is a proprietary audio/video interface for transmitting uncompressed video data and compressed or uncompressed digital audio data from an HDMI-compliant source device to a compatible display device. It is commonly used for connecting devices like TVs, monitors, and projectors.

4 C++

4.1 Classes

- **Class Definition**
  - Syntax and structure
  - Access specifiers: public, private, protected
- **Inheritance**
  - Single and multiple inheritance
  - Virtual inheritance
- **Polymorphism**
  - Function overloading
  - Operator overloading
  - Virtual functions and abstract classes

4.2 Containers

- Sequence
  - **array**

```
std::array<int, 3> arr; // uninitialized (whatever was in memory before)
std::array<int, 3> arr = {}; // initialized as 0s
std::array<int, 3> arr1 = {1, 2, 3};
std::array<int, 3> arr2{1, 2, 4};
arr1.fill(0); // fills array with 0s
arr1.swap(arr2); // swaps contents of arr1 and arr2
```

- **vector**

```cpp
std::vector<int> v;
v.capacity(); // size of currently allocated memory
v.shrink_to_fit(); // releases unused memory
v.reserve(100); // pre-allocates 100 elements
v.clear(); // erases all elements
v.erase(v.begin()); // erases first element
v.push_back(1); // adds 1 to the end
v.rbegin(); // reverses iterator
std::erase_if(v, [](int x) { return x > 10; }); // removes all
    elements > 10
std::vector<Pair<int,int>> classV;
classV.emplace_back(10,1); // create Pair object and push to
    back
```

- **inplace_vector**
- **deque**

• Associative

- **Set**
- **Map**
- **Multiset**
- **Multimap**

• Unordered Associative

- **unordered_set**
- **unordered_map**
- **unordered_multiset**
- **unordered_multimap**

• Adaptors

- **stack**
- **queue**
- **priority_queue**
- **flat_set**
- **flat_map**
- **flat_multiset**
- **flat_multimap**

4.3 Modern C++

• C++11

- **Alias Templates**
- **atomic**
  Well-defined behavior in the event of RMW race contition. Accesses to atomics may establish inter-thread synchronization and order non-atomic accesses.

```cpp
atomic_bool b; // same as std::atomic<bool> b;
```

- **auto**
- **constexpr**

- **final**
  - Specifies that a class cannot be inherited from.
  - When used in a virtual function, specifies that the function cannot be overridden by a derived class.
  - final is also a legal variable/function name. Only has special meaning in member function declaration or class head.

```cpp
struct Base
{
    virtual void foo();
};
struct A : Base
{
    void foo() final; // Base::foo is overridden and A::foo is the final override
    void bar() final; // Error: bar cannot be final as it is non-virtual
};

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B {}; // Error: B is final
```

- **initializer list**
- **iota**

```cpp
void iota(ForwardIterator begin, ForwardIterator end, T v); // fills range [first-last] with sequentially increasing values starting at v in begin
```

- **lambdas**

  capture comma-separated list of variables which are captured/modified by the lambda. Captures cannot have same name as input parameters.

  Capture list
  - & = capture all used variables by reference
  - = = capture all used variables by copy
  - varName = by-copy
  - varName... = by-copy pack-expansion
  - varName initializer = by-copy w/ initializer
  - &varName = by-reference
  - &varName... = by-reference pack-expansion
  - &varName initializer = by-reference w/ initializer
  - this = by-reference capture of current object
  - *this = by-copy capture of current object
  - ... = by-copy capture of all objects w/ pack expansion
  - &... initializer = by-reference w/ initializer and pack expansion

```cpp
// If the capture-default is &, subsequent simple captures must not begin with &.
[&] {};          // OK: by-reference capture default
[&, i] {};       // OK: by-reference capture, except i is captured by copy
[&, &i] {};       // Error: by-reference capture when by-
    reference is the default
[&, this] {};     // OK, equivalent to [&]
[&, this, i] {}; // OK, equivalent to [&, i]
```

```
// If the capture-default is =, subsequent simple captures
    must begin with & or be *this(since C++17) or this(since C
    ++20).
[=] {};          // OK: by-copy capture default
[=, &i] {};      // OK: by-copy capture, except i is captured by
     reference
[=, *this] {}; // until C++17: Error: invalid syntax
               // since C++17: OK: captures the enclosing S2
                  by copy
[=, this] {};  // until C++20: Error: this when = is the
    default
                // since C++20: OK, same as [=]
```

```
```

- **mutex**
- **override**
- **random**

```
#include <stdlib>
int rand(); // returns integer in [0, RAND_MAX]
```

```
#include <random>
// default_random_engine
// philox4x64 -> philox_engine
// random_device = non-deterministic generator based on
    hardware entropy
std::random_device rd;
rd.entropy(); // estimate of random number device entropy.
    Deterministic entropy = 0.
std::uniform_real_distribution<double> dist(0.0, 1.0);
```

Distribution list

- uniform
  - int
  - real (double)
- bernoulli
  - bernoulli
  - binomial
  - negative binomial
  - geometric
- Poisson
  - poisson
  - exponential
  - gamma
  - weibull
  - extreme_value
- Normal
  - normal
  - lognormal
  - chi_squared
  - cauchy

            - fisher_f
            - student_t
        - Sampling
            - discrete
            - piecewise_constant
            - piecewise_linear
            - item4
    - **range-based for**
    - **thread**
    - **trailing return type**

- C++14
    - **Variable Templates**
    - **Generic Lambdas**

- C++17
    - **tuple**
    - **execution policies**

- C++20
    - **Modules**
    - **Coroutines**
    - **Ranges**
      Extension/Generalization of algorithms and iterator libraries to make them less error-prone. Ranges are an abstraction of the following:
        - [begin, end) iterator pair : ranges::sort()
        - begin + [0, size) : views::counted()
        - [begin, predicate) : views::take_while() (conditionally-terminated sequences
        - [begin, ..) : unbounded (e.g. views::iota())

      std::views // shorthand for std::ranges::views TODO: do more usage/investigation on these
    - **Midpoint**
      Can be used on any arithmetic type, excluding bool. Can be used on objects as long as they are not incomplete types. Returns half the sum of the two inputs, no overflow occurs (this is the main reason to use STL rather than custom implementation). Inputs must point to elements in same object, else behavior is undefined. In case of decimal in average, rounds down.
    - **using enum**
    - **constinit**
    - **string formatting**
    - **template concepts**
    - **coroutines**
    - **modules**

- C++23
    - **print/println**

```cpp
#include <print>
std::print("{0} {2}{1}!", "Hello", 23, "C++"););
std::println(); // adds newline to std::print();
```

- **byteswap**

```cpp
#include <bit>
std::byteswap(T n) noexcept; // T can be any integer value
```

- **flat_map/flat_set**

4.4 Concepts

- **Types**
- **RAII**
- **item3**
- **item4**