Concurrent Programming in C++

Contents

1 Introduction

1.1 Motivation

Traditionally, software has been written to execute sequentially, i.e., one step at a time. The execution proceeds to the next step only after executing the current step. Hence, the program's runtime is limited by how fast each instruction can be executed. The speed to execute an instruction depends on the processor frequency and the speed of accessing resources like memory, user IO, database, etc. If the speed of executing an instruction is more than the speed of accessing a resource, the processor is idle while waiting for resource access. In this case, the processor's capacity is not fully utilized.

Further, most modern computers have multicore processors. Sequential programs run on a single core and, thus, don't utilize the full capacity of the hardware.

Suppose, the programs are written in a way that some of them can run separately (independently), then a higher processor capacity can be utilized by running different parts of the program either during a part's idle time or on multiple cores. **The goal of concurrent and parallel programming is to reduce a program's runtime by utilizing the available compute resources to a higher extent.**

1.2 Concurrent vs Parallel

Concurrency is the ability of a program to be broken into parts such that they can be executed in fully or at least partially independently *without affecting the end result*. It is about how a program is structured and the composition of the independently executing parts.

Parallelism is about simultaneously executing multiple things at once while concurrency is about dealing with multiple things at once. A concurrent program allows for parallelism given the hardware but it need not necessarily be parallel. For a program to actually execute in parallel, relevant hardware is needed like an extra core or a processor.

Concurrent tasks need not always be executing parallelly. When accessing the data is slower than executing an instruction on it, the overall speed of a sequential program is limited by the speed of the data access. If a program is 'concurrently structured', then during the wait time of accessing the data, other instructions can be executed. For e.g., all IO devices run concurrently but they can all run on a single processor because each device's execution frequency is much lower compared to the processing frequency. Hence, they can take turns on a single processor and still appear to run in real time to the user. **Concurrency is a solution to the reduced processing speed because of IO bound operations.**

1.3 Processes vs Threads

When an instance of an application is fired up, a process is created by the OS. A process consists of the program's code, data, and the state information. Each process is independent and has its own address space in the memory. Hence, multiple processes of the same program can run independently.

OS is responsible to manage all the processes. Each process can further have threads, which are light-weight processes (LWP). Threads share the same address space and code, run independently, and exist only as a part of a process.

A thread is instantiated as follows: `std::thread th(concurrent_fn, arg_1, .., arg_n)`. The thread `th` executes `concurrent_fn` with its arguments as `arg_1` to `arg_n`.

Just because a program is structured with multiple threads, does not mean it will be executed in parallel. Multiple threads can take turns on the same core (switching) or they can be executed on multiple cores (parallel).

Since threads share the same address space, sharing data among them is easy, for e.g., with global variables. However, sharing data between processes is comparitively difficult because every process exists in its own address space and Inter Process Communication (IPC) is necessary.

Programs can be written to use either multiple process or multiple threads or a combination of both. **A rule of thumb is threads are preferable if the program can be structured to use them.** Threads are more efficient and faster than processes because

- threads have a lesser overhead to create and terminate
- threads are faster for the OS to switch
- communication between threads is easier

However, the best choice depends on what is to be done and the environment it is running in because the implementation of processes and threads differ with OS and language.

1.3.1 Thread Life Cycle

When a new process or program begins, the main thread is created. It can start additional child threads which can further spawn more threads. All these threads belong to the same process but they can be executed independently. Threads can be in one of the following four states:

- New: When the thread is created
- Runnable: OS can schedule the thread to execute. Can come either from new or blocked.
- Blocked: If the thread is waiting for something (IO or for another thread to be done), runnable moves to blocked. No compute is used.

- Terminated: When the thread's job is done or abnormally aborted, it enters terminated from runnable.

While C++ does not provide the thread state, it can be checked if the thread is still considered active or not with `joinable()`.

1.4 Memory Image: Single- vs Multi-threading

A sequential program runs on a single thread. Its memory image has program code, stack, and heap. The program counter (PC) register points to the line of code that is being executed. Stack pointer (SP) points to the stack frame of the current function call.

In multi-threaded programs, multiple lines can be executed and hence there is a PC for each thread. Each thread also has a seperate stack pointer. Each thead makes different function calls and may make multiple calls to the same function. Hence, the local stack variables should be independent to be able to run the same function multiple times with different data. Hence, each thread has its own SP. However, the heap memory and global variables are shared among the threads.

1.5 'Hello Concurrent World!'

```cpp
#include <thread>
#include <chrono>

void thread_job() {
    printf("Hello from thread_job() running on thread %x.\n",
        std::this_thread::get_id());
    printf("Start of 3 sec wait in thread_job().\n");
    std::this_thread::sleep_for(std::chrono::seconds(3));
    printf("End of 3 sec wait in thread_job().\n");
}

int main() {
    printf("Hello from main() running on thread %x.\n",
        std::this_thread::get_id());
    std::thread T1(thread_job);

    printf("Waiting for T1 to complete its job.\n");
    T1.join();
    printf("Job done.\n");

    return 0;
}
```

By default, a thread is created to execute `main()`. This thread (parent) creates a thread `T1` (child) in line 15, which executes `thread_job()`. While creating a thread, a job must also be assigned to it. The parent thread waits for the child thread to complete its execution at line 18.

1.6 Execution Scheduling

A scheduler in an OS decides which thread and when a thread is allotted processing resources. When a process is created and ready to run, it is loaded into the memory and placed in the ready queue. Scheduler allots a processor to these processes. If a process has to wait, then it is moved into a different queue where it waits and then another process uses the processor. Changing of processes is called **context switching**.

The OS stores and retrives the state/context of the process from a process control block (PCB) and of a thread from thread control block (TCB). Context switching takes sometime to save the state of an outgoing process and load the state of an incoming process. Hence, the scheduler has a strategy called scheduler algorithms to perform these switches. An OS can choose to schedule the process/threads differently from run to run. Hence, **concurrent programs should be robust to order and time of execution of the threads/processes**, which pose synchronization challenges. **Programs should not rely on execution scheduling for correctness.**

1.6.1 Kernel- and User-level Threads

Threads that are scheduled independently by the kernel are called kernel threads, for e.g., Linux pthreads. In constract some libraries provide, user-level threads that a multiplexed over a smaller number of kernel threads. There is a lower overhead in switching user threads but they can't run parallely.

1.7 Synchronization Challenges

One of the main challenges with writing concurrent programs is identifying the dependencies between threads and handling them properly. Mishandling the dependencies can lead to a non-deterministic behaviour of the program. Further, an unexpected result mayn't occur everytime , which pose challenges in reproducing it behaviour. Hence, **pay a super-close attention when multiple threads are spawned and access the same resource concurrently**.

1.7.1 Data Race

A 'data race' can occur if two or more threads concurrently access the same memory location and at least one of those is writing to that location. It can be avoided with synchronization techniques. Following program demonstrates a data race for `counter` variable:

```
#include <thread>
```

```
uint counter = 0;
constexpr uint thread_count_limit = 1e6;

void increment_counter() {
    for (uint i = 0; i < thread_count_limit; i++) counter++;
}

int main() {
    std::thread T1(increment_counter);
    std::thread T2(increment_counter);

    T1.join();
    T2.join();

    printf("Expected = %d, Actual = %d, Error = %d\n",
        2 * thread_count_limit, counter,
        2 * thread_count_limit - counter);

    return 0;
}
```

the printed output looks as follows:

```
Expected = 200000, Actual = 126857, Error = 73143
```

The `counter` variable is incremented for `thread_count_limit` times by each of the threads, `T1` and `T2`. However, the final value of the counter is much lesser than the expected value. It happens so because of concurrent access of the counter by both the threads to increment it. Incrementing a counter is not an atomic operation in assembly code, rather it is the following 3 steps:

1. moving the counter into a register
2. increment the value in register by 1
3. write the value from register into the counter

Let's say the counter is 100 and a context switch happens when `T1` completed step 2 but before step 3. The register value is 101 but the counter value is still 100. Say `T2` is active and it still sees the counter as 100 even though the increment is completed, which also increases the counter to 101. Now, say, `T2` is blocked after the 3 steps given above and `T1` is again made active. `T1` continues execution from where it was blocked and it writes 101 to the couter. Although the increment happened twice, effectively only one increment is reflected. Such a behaviour can occur multiple times resulting in the observed error.

A data race occurs when a thread is modifying a value in between another thread reading and writing to the it.

Further, the actual counter value most probably is different in different runs – non deterministic behaviour due to data race. Also, if the value of `thread_count_limit` is lower, then the chances of concurrent memory access is also lower. As a result, for a low enough `thread_count_limit`, the actual counter value can be the same as the expected value. However, it does not imply that the program is free of data race but means that a concurrent data access has not manifested in that particular run. Such behaviour makes reproducing and fixing data race bugs very challenging.

1.7.2 Race Conditions

A race condition is a flaw in the ordering or timing of a thread's execution that causes incorrect behaviour. Race condition leads to data races and vice versa. But both can exist without each other as well. This is a super-challenging bug to find because once a debugger is turned on, the timing no more remains the same and the bug may not reoccur. The following example demonstrates a race condition.

```cpp
#include <thread>

uint counter = 0;

void increment_counter() {
    counter++;
}

void double_counter() {
    counter *= 2;
}

int main() {
    std::thread T1(increment_counter);
    std::thread T2(double_counter);

    T1.join();
    T2.join();

    printf("Expected = %d, Actual = %d, Error = %d\n",
        2, counter, 2 - counter);

    return 0;
}
```

It is intended to increment `counter` by 1 in a thread and then double it in another thread. With an initial value of 0, the expected result is 2. However, incrementing and doubling may not occur in the same order as expected because `T2` can be scheduled after `T1`, where doubling occurs first and then incrementing that results in 1. It has to be noted that a data race also exists in the above code.

Next two sections are about how to avoid data race and race conditions by using synchronization techniques so that concurrent read and write access is avoided and the threads execute the instructions in the desired order irrespective of their scheduling order.

1.8 Summary

1.8.1 Concepts

- Concurrency is the ability of a program to be broken into parts to be executed in fully or at least partially independently without affecting the end result.
- Concurrency is a solution to the limited speed of serial processing due to IO bound operations. When a part of the program waits for IO operations, other parts can be executed on the same processor.
- Parallelism is about simultaneously executing multiple things at once while concurrency is structuring the program in a way that multiple things can be dealt with at once.
- Parallel processing needs multiple processors while concurrency is possible on a single processor also.
- Processes have their own address space while multiple threads of the same process share the address space. Hence, threads are faster to create and terminate and the OS can quickly switch them, which results in threads being efficient and faster than processes. Threads are also referred to as Light Weight Processes (LWP).
- As a rule of thumb, if a problem can be solved with multiple threads, then choose it over multiple processes.
- A scheduler in an OS decides which thread and when it is allotted processing resources. It can choose to schedule the threads differently from run to run. Hence, concurrent programs should not rely on order and time of execution of the threads for correctness. Otherwise, it leads to a ‘race condition’.
- A ‘data race’ occurs if two or more threads concurrently access the same memory location and at least one of those writing to that location.
- A ‘race condition’ occurs when the ordering or scheduling of threads impacts the end result.
- Fixing concurrency bugs is super hard. It is, of course, desirable to avoid such issues in the first place. Pay a super-close attention whenever two or more threads access the same resource.

1.8.2 C++ syntax

- Use `-pthread` as a compiler flag to compile multi threaded programs.
- `std::this_thread::get_id()` gets the ID of the thread in which it is called.
- `std::this_thread::sleep_for(std::chrono::seconds(1))` puts the thread to sleep for 1 sec at this instruction.
- `std::thread th(concurrent_fn, arg_1, .., arg_n)` instantiates a thread `th` that runs the `concurrent_fn` with arguments `arg_1, .., arg_n`.
- The parent thread waits for the child thread where the child's `join()` method is called.
- `joinable()` returns `true` if the thread is not in terminated state.

2 Synchronization: Mutual Exclusion (Mutex)

When a critical section can be prohibited from multiple threads accessing it at once with atleast one thread writing to it, a data race can be prevented. Consequently, only one thread reads/writes to it at a time, which can be achieved with a Mutex. `std::mutex` has `lock()` and `unlock()` methods with which the critical section can be protected.

When a thread locks a mutex, the critical section (set of instructions and data the mutex guards) is only accessible to that thread and the other threads are prohibited from accessing the critical section during that time.

Only a single thread can lock a mutex at a time. Locking a mutex is an atomic operation, although it takes multiple steps and non zero time to lock it. If a thread is locking it, another thread can't lock it. So either a thread acquires a mutex or not. No other intermediate state is possible. Consequently, a thread can lock a mutex only in an unlocked state.

Data race from a previous example is fixed with a mutex as follows:

```cpp
#include <thread>
#include <mutex>

uint counter = 0;
uint thread_count_limit = 1e5;
std::mutex mtx;

void increment_counter() {
    for (uint i = 0; i < thread_count_limit; i++) {
        mtx.lock();
        counter++;
        mtx.unlock();
    }
}

int main() {
    std::thread T1(increment_counter);
    std::thread T2(increment_counter);

    T1.join();
    T2.join();

    printf("Expected = %d, Actual = %d, Error = %d\n",
        2 * thread_count_limit, counter, 2 * thread_count_limit - counter);

    return 0;
}
```

the printed output looks as follows:

```
Expected = 200000, Actual = 200000, Error = 0
```

If a thread attempts to lock an already locked mutex and it waits until the mutex is unlocked. Hence, if a mutex is not unlocked then the a certain thread can wait forever leading to to the program never completing the execution.

Threads, while waiting for a locked mutex, may be idle. Hence, it is important to structure the critical section only to perform the necessary operations. Otherwise, it could lead to unnecessary increase in the run time of the entire program.

2.1 Non-blocking locking

`try_lock()`: It may not be efficient for a thread to wait forever for the mutex when it is locked. Locks the mutex if it is unlocked. Else the control steps ahead to perform some other work.

2.2 Timed mutex

Gives control to limit the time of attempting to lock the mutex to a finite value.

- `try_lock_for()`: Waits atmost for the specified time duration to lock the mutex.
- `try_lock_until()`: Waits atmost till the specified absolute time to lock the mutex.

2.3 Recursive/reentrant Mutex

If a mutex is locked successively by the same thread, then the program enters a dead-lock situation and it never ends. Since the mutex is already locked by the thread and when the second lock is called by the it, it waits forever assuming that it can lock the mutex the second time when the mutex is unlocked. But the mutex can't be unlocked because the thread that can unlock it is waiting to lock the mutex again which leads to 'deadlock'. Usually, locking the same critical code multiple times should not be necessary. But during recursive calls, such multiple locks of the same code can occur, when a recursive lock has to be used to avoid a never-ending behaviour. A recursive mutex has to be unlocked as many times as it is locked, for a different thread to be able to lock the mutex again.

2.4 Shared Mutex

It's alright for multiple threads to access the same memory location as long as all of them read from the same location. With the `std::mutex`, if one thread locks for reading, then the other thread can't read the memory although that operation is safe. This can be solved with a 'Reader-Writer' lock. It can be locked in either of two modes

- shared read: allows multiple threads to only read it to lock it
- exclusive write: only one thread at a time can lock it

Exclusive write access can be granted only when no thread has the shared read access. Don't use it always because it uses more resources under the hood for reader counting. As a rule of thumb, it makes sense to use a shared mutex, if the number of readings/readers is more than writings/writers.

Other hybrid mutexes like, recursive timed mutex and shared timed mutex are also available that provide a hybrid functionality.

2.5 Mutex Wrapper

2.5.1 `lock_guard`

Mutex, like heap memory, is a dynamically-managed resource. Hence, the best practice is to use it in an RAII-style mechanism. A `lock_guard` realizes RAII for a mutex. When a `lock_guard` object is created, it attempts to take ownership of the mutex and locks it. When the control leaves the scope in which the `lock_guard` object was created, the `lock_guard` is destructed and the mutex is unlocked.

The `increment_counter()` from the above example, where a raw mutex is used, is modified to use a `lock_guard`.

```
1 void increment_counter() {
2     std::lock_guard<std::mutex> lg(mtx);
3     for (uint i = 0; i < thread_count_limit; i++) {
4         counter++;
5     }
6 }
```

`mtx` is locked in line 2. It is unlocked in line 5 when `lg`'s destructor is called. A `lock_guard` object can't be explicity locked or unlocked.

2.5.2 `unique_lock`

It is a general-purpose mutex wrapper with a larger set of functionalities as follows:

- Multiple locking strategies by passing the following arguments to the constructor
  - `defer_lock`: do not lock the mutex object automatically on construction
  - `try_to_lock`: attempts to lock the mutex object by calling its `try_lock` member instead of its `lock` member
  - `adopt_lock`: assume that the mutex is already locked by the current thread
- Locking attempt with a time limit
- Recursive locking
- Transfer of lock ownership using move
- Supports condition variables

2.5.3 `scoped_lock`

Similar to `lock_guard` but can additionally act as a wrapper to multiple mutexes. This helps in avoiding concurrency bugs that occur due to ordering of locking and unlocking of mutexes in multiple threads and ofcourse also provides RAII-mechanisms for those mutexes.

2.6 Mutex Implementation

Lock implementation should satisfy:

- Mutual exclusion
- Fairness: All threads should eventually get the lock. No starvation.
- Low overhead: For acquiring, releasing, and waiting should not consume a lot of resources.

Implementing locks needs support from both userspace programs and kernal code.

Implementing a mutex by using only software can be done by disabling interrupts for locking and then enabling them for unlocking. However, it is very fundamental aspect of the OS and the bugs in user code can cause a larger issue. Further, in multi core processors, the interrupts on other processors can be active. Hence, risky and limited application.

Hence, support is needed from the hardware as well. Test-and-set instrutions are executed atomically by the hardware. Hence, checking for a flag and then setting it can be done together. Without hardware support, data race from the code is simply transfered to a data race in the mutex flag. Further kinds of hardware support exists like, compare-and-swap.

Spinlock mutex: If a thread can't get a lock, it spins until the lock is obtained.

Sleeping mutex: Another option is to yield, i.e., put itself in a blocked state if the lock is not obtained. Scheduler can activate it again when it can check for the mutex availability.

Most userspace implementations are sleeping mutex. Locks inside the OS are always spinlocks. OS can't sleep because if it sleeps, no one else can wake it up. Default software layer that runs. It can't yield to anything. Important to avoid deadlocks while using spinlocks.

2.7 Summary

2.7.1 Concepts

- A Mutex allows to safely access a shared resource by restricting write access to one thread at a time. It is used to lock 'critical code' with the shared resource where mutual exclusion is necessary.
- Locking a mutex is an atomic operation. A thread can either lock it or not. A thread that locks it, has the access to the shared resource. No other thread can access the resource until the thread that acquired the mutex unlocks it.
- All types of mutexes provides `lock()` and `unlock()` methods. A mutex that is locked must be unlocked by the same thread to avoid concurrency bugs.
- Several mutexes are available with an additional functionality
  - `timed_mutex`: Attemps to `lock()` for a finite time. If the lock can't be obtained, the control proceeds ahead.
  - `recursive_mutex`: Can be locked and unlocked multiple times but the same number of times.
  - `shared_mutex`: Can be read by multiple threads, however, written by only one thread.
  - `recursive_timed_mutex` and `shared_timed_mutex`: Hybrid of the above mutexes.
- A mutex wrapper provides an RAII-style mechanism for a mutex, for e.g., `lock_guard`, `unique_lock`, and `scoped_lock`. Use a mutex with a wrapper like any other dynamically managed resources to reduce the focus on mutex management.
  - `lock_guard`: Obtains the lock on mutex with the object is created and unlocks when it goes out of scope. Use this if no other functionality is needed.
  - `unique_lock`: General purpose wrapper with a larget set of functionalities to support multiple types of mutexes, transfering ownership, and condition variables.

    - `scoped_lock`: Similar to `lock_guard` but can additionally act as a wrapper to multiple mutexes and consequently avoids a deadlock that can occur due to usage of multiple mutexes.

- A mutex needs support from the hardware as well, where the hardware excutes a sequence of instructions atomically. Test-and-set instructions and Compare-and-swap help in achieving it.

2.7.2 C++ syntax

- To create a mutex object: `std::mutex mtx;`
- `lock()` allows the access of a section for a single thread that acquired the mutex until `unlock()` is called.
- `try_lock()` returns true and acquires lock if possible else returns false

3 Synchronization: Waiting and Signalling

Consider implementing a shared queue where multiple threads are putting in and are also taking out items. A thread that is adding an item should not add to a full queue and similarly a thread trying to remove an item should not remove from an empty queue. Although a mutex helps to safely add and remove the items, there is also a need for the threads to wait when the queue is either empty or full and signal others when its is not so. A condition variable provides a waiting and signalling mechanism for the threads to work synchronously.

3.1 Condition Variable

3.1.1 Motivation

Consider the below example where `process_data()` should compute the result only after the `generate_data()` has appropriately filled `data_.value`. Since both of them run on different threads, a synchronization mechanism is necessary that ensures `generate_data()` executes before `process_data()` irrespective of the order in which the threads are scheduled.

```cpp
#include <thread>
#include <condition_variable>
#include <chrono>

struct data {
    int value{0};
    bool status{false};
};

```

```cpp
std::mutex mtx_;
data data_;

void generate_data() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::unique_lock<std::mutex> ul(mtx_);
    data_.value = 10;
    data_.status = true;
    ul.unlock();
}

void process_data() {
    int result;

    std::unique_lock<std::mutex> ul(mtx_);
    while (true) {
        if (data_.status == true) {
            result = 3 * data_.value;
            break;
        } else {
            ul.unlock();
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
            ul.lock();
        }
    }
    ul.unlock();
    printf("Result is: %d\n", result);
}

int main() {
    std::thread generator(generate_data);
    std::thread processor(process_data);

    generator.join();
    processor.join();
    return 0;
}
```

Since `process_data()` has no idea about when `generate_data()` will fill the value, it executes a while loop in which it checks if the data is generated or not. To check the condition, the mutex is locked. If the condition is not met `generate_data()` didn't generate the value yet. Hence `processor` thread unlocks(), waits for sometime, allowing the generator to obtain the lock and then locks again to check the condition. Until the condition is met, `processor` thread is performing unproductive work of locking the mutex, checking the condition, unlocking the mutex every cycle thereby consuming processor time and reduding the availability of mutex leading to reduced productivity.

In a general scenario when a thread can only start its work after a condition (that changes due to multiple threads) is met or when another thread has completed a job, a synchronization among the threads is necessary for the program to be efficient. Such a synchronization is possible with a condition variable. It provides a mechanism for the threads to wait and signal each other.

Without a condition variable, the thread has to continuously check the condition and proceed ahead when it is met. However, this unnecessary condition checking consumes processing resource. Such a behaviour is called as spinning (busy waiting) and it is inefficient when it occurs frequently. Spinning is generally bad because the thread itself is unproductive and further reduces the productivity of other threads.

3.1.2 Usage

A condition variable provides some of the following key methods that help to avoid spinning:

1. wait (`wait()`)
2. signal (`notify_one()`)
3. broadcast (`notify_all()`)

`wait(lock, <optional> predicate)`: It atomically unlocks the mutex and puts the caller thread to sleep until notification/wake up call arrives. An optional predicate can be provided that checks for a condition associated with the condition variable. When a wake up call arrives and if the condition is satisfied, the wait exits and the lock is reacquired.

`notify_one()`: When a thread is done with its job, it notifies/wakes up a waiting thread.

`notify_all()`: It is similar to the `notify_one()` operation but it wakes up all of the threads in the waiting queue.

The following example modifies the above example to use a condition variable that results in synchronization without spinning:

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

struct data {
    int value{0};
    bool status{false};
};
```

```

std::mutex mtx_;
std::condition_variable cv_;
data data_;

void generate_data() {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::unique_lock<std::mutex> ul(mtx_);
    data_.value = 10;
    data_.status = true;
    ul.unlock();
    cv_.notify_one();
}

void process_data() {
    int result;

    std::unique_lock<std::mutex> ul(mtx_);
    cv_.wait(ul, [](){return data_.status;});
    result = 3 * data_.value;
    ul.unlock();
    printf("Result is: %d\n", result);
}

int main() {
    std::thread generator(generate_data);
    std::thread processor(process_data);

    generator.join();
    processor.join();
    return 0;
}
```

3.1.3 `wait()` under the hood:

The lock associated with the condition variable must be acquired before entering `wait()`. If `wait()` is called the first time, the thread locks the mutex and then checks the predicate. Else, when it gets a wake up call while sleeping, the thread becomes active, reacquires the lock, and checks the predicate. In both scenarios, if the predicate call returns

- true, then the thread continues its work
- false, then `wait()` unlocks the mutex and puts the thread to sleep

Since the condition is checked by locking the mutex when a notification/wake up arrives, the operation is both safe from data races and race conditions. In case of a spurious wake up call, the condition is not satisfied and the thread is put to sleep again. In case of a lost wake up call, the condition is not satisfied and the thread does not wait at all. The condition acts as a memory for the condition variable's wait and notify mechanism.

The usage of predicate in the condition variable is equivalent to the following code:

```
std::unique_lock<std::mutex> lk(cv_m);

while !(predicate_return) {
  cv.wait(lk);
}
```

Reference

1. C++ Core Guidelines: Be Aware of the Traps of Condition Variables

3.1.4 Avoiding a race condition

Even if the data type involved in the predicate is atomic and the predicate check only reads the atomic data, a mutex is still necessary to avoid a race condition. To demonstrate it, consider the following execution sequence from the above example:

- predicate_return is false when it is checked to enter the while loop, then
- predicate_return becomes true and a notification arrives, and then
- `wait()` is executed and the thread is put to sleep

No more notifications come and the thread sleeps forever leading the thread waiting forever (deadlock). The bug is a result of order of execution of the threads and, hence, it a race condition. The notication action by a thread is sandwitched between the actions of the waiting thread. However, by using a mutex, it can be ensured that either the waiting thread or notifying thread executes in not an intertwined manner as above, which is achieved using a monitor construct.

3.1.5 Monitor

Condition variable is a queue for threads that are put to blocked state. These threads wait for a signal to become unblocked. It is associated to a mutex and works together with a mutex to implement a higher level construct called monitor. Without a mutex, as seen above, a race condition is possible that leads to a deadlock.

Monitor: Protects a section of code with mutual exclusion and provides the ability for threads to wait until a condition is met. Monitor is analogous to a room with the procedures and shared data. Only one thread can be in that room at a time and, hence, only it can access the instructions and resources at that moment. Mutex is like a lock on the door.

Threads that try to enter the room while it is occupied wait outside in the condition variable's queue. Multiple threads could be waiting for the same condition to occur or for multiple conditions. A mutex can be associated with multiple condition variables.

3.2 Producer-Consumer Pattern

A common design pattern in concurrent programming is where one or more threads add elements to a data structure and some other threads remove elements from it. Several synchronization challenges exist as follows:

1. Mutual exclusion of producers and consumers
2. Must not add data when full
3. Must not remove data when empty

Some languages provide a thread-safe, synchronized, data structures. If they are not available, then a mutex and a condition variable can be used together to implement thread-safe data structures. A thread safe queue is implemented in the following example:

```
template <typename type>
class ThreadSafeQueue {
public:
    void push(type data) {
        std::unique_lock<std::mutex> ul(mtx_);
        buffer_.push(data);
        ul.unlock();
        if (buffer_.size() == 1) cv_.notify_one();
    }

    type pop() {
        std::unique_lock<std::mutex> ul(mtx_);
        while (buffer_.empty()) {
            cv_.wait(ul);
        }
        type data = buffer_.front();
        buffer_.pop();
        ul.unlock();
        return data;
    }

private:
    std::queue<type> buffer_;
    std::mutex mtx_;
    std::condition_variable cv_;
};
```

The above implementation is based on the monitor construct. Either `push()` or `pop()` can access the shared resources and the instructions at a time. When the buffer is empty, the thread that accesses `pop()` is blocked and put into `cv_`'s waiting queue. Once some other thread pushes data into the shared buffer, the size increases from 0 to 1 that results in a notification. With the arrival of a notification, a waiting thread is unblocked and it can pop the element.

The above implemention of a thread-safe queue is used to synchronize production and consumption as follows:

```
ThreadSafeQueue<int> buffer_;

void producer(uint start, uint end, uint &num_produced) {
    for (uint index = start; index < end; index++) {
        buffer_.push(index);
        num_produced = index - start + 1;
    }
}

void consumer(uint &num_consumed) {
    num_consumed = 0;
    while (true)
    {
        int data = buffer_.pop();
        if (data == -1) {
            buffer_.push(data);
            return;
        } else {
            num_consumed++;
        }
    }
}

int main() {
    constexpr uint num_producers = 6, num_consumers = 9;
    std::vector<std::thread> producer_vec;
    std::vector<std::thread> consumer_vec;
    std::array<uint, num_producers> num_produced;
    std::array<uint, num_consumers> num_consumed;
    for (uint index = 0; index < num_producers; index++) {
        uint start = 50 * index;
        uint end = 50 * (index + 1);
        std::thread th(producer, start, end,
            std::ref(num_produced[index]));
        producer_vec.push_back(std::move(th));
    }

    for (uint index = 0; index < num_consumers; index++) {
        std::thread th(consumer, std::ref(num_consumed[index]));
        consumer_vec.push_back(std::move(th));
    }

    for (std::thread &th : producer_vec) {
        th.join();
    }
    // All threads are done producing.
    // Adding an end condition to signal the consumers to stop.
    buffer_.push(-1);

    for (auto &th : consumer_vec) {
        th.join();
    }

    uint total_produced, total_consumed;
    total_produced = std::accumulate(num_produced.begin(),
        num_produced.end(), 0);
    total_consumed = std::accumulate(num_consumed.begin(),
        num_consumed.end(), 0);

    printf("total_produced = %d, total_consumed = %d\n",
        total_produced, total_consumed);
    return 0;
}
```

With an upper limit on the buffer size, if the producer is faster than consumer, an overflow occurs leading to a data loss. Some unlimited size data structures exist but they too are eventually limited by the physical memory. In general, average rate of production should be less than the average rate of consumption to avoid data loss. If multiple tasks are to be performed, the same producer-consumer setup can be extended into a pipeline. A consumer in the prior task is the producer to the next task leading to Multiple producer consumer pairs that are connected by a buffer like data structure.

3.3 Semaphore

A semaphore is another synchronization mechanism to control access to shared resources.

- It can allow multiple threads to access the resource at the same time.
- A pre-defined maximum number of threads can access it.
- It includes a counter to track how many threads acquired it.

It offers the following two important methods:

- `acquire()`:
  - The counter is at the maximum value when no thread acquired it.
  - When a thread acquires it, the counter goes down by 1.
  - The threads trying to acquire the semaphore when the counter is 0 are blocked and wait in the semaphore's queue.
- `release()`:
  - When a thread releases it, the counter goes up by 1.
  - The waiting threads are signalled to wake up.

A semaphore that can be acquired by more than one thread at a time is called a counting semaphore. Otherwise, it is called a binary semaphore. A common application of a counting semaphore is to track a limited resources like number of connections available to a data base, number of elements in a queue, e.t.c.

The differentiating aspect of a semaphore is one thread can acquire it and another thread can release it. It is not necessary that the thread that acquires it releases it for the semaphore counter to change.

A semaphore is implemented using a mutex and a condition variable as follows:

```
class Semaphore {
public:
    Semaphore(uint count, uint max_count)
        : count_(count), max_count_(max_count) {}

    void acquire() {
        std::unique_lock<std::mutex> ul(mtx_);
        while (count_ == 0) {
            cv_.wait(ul);
        }
        count_--;
        if (count_ == max_count_ - 1) cv_.notify_all();
    }
    void release() {
        std::unique_lock<std::mutex> ul(mtx_);
        while (count_ == max_count_) {
            cv_.wait(ul);
        }
        count_++;
        if (count_ == 1) cv_.notify_all();
    }

private:
    uint count_ = 0, max_count_ = 0;
    std::mutex mtx_;
    std::condition_variable cv_;
};
```

3.3.1 Produce-Consumer Semaphore

A pair of semaphores can be used to push and pop data from a shared queue. One semaphore tracks the number of items and the other tracks the number of free spaces.

TODO: Use a boost semaphore for producer-consumer problem

To add an element to the buffer, the producer atomically acquires the empty count semaphore (which reduces it by one) and pushes data into the buffer and releases the fill count semaphore (which increments it by 1). Now a consumer thread, first acquires the fill count semaphore (which reduces it by 1) removes the item from the buffer and releases the empty counter semaphore (which increases it by 1). Both the producer and consumer, acquire and release different locks.

If the consumer tries to remove an element when the fill count semaphore is 0, then it is put to wait and is signalled once a consumer fills the data and releases the fill counter. Similarly, if the empty counter semaphore is zero, the producer is put to wait and is signalled once a consumer removes an item and releases the empty counter.

3.4 Barriers

It lets the threads to wait until a given number of threads have come to wait and then releases all of them. It allows to synchronize the actions among the threads irrespective of the order in which they are scheduled by the OS.

A mutex and a condition variable are used together to implement a barrier as follows:

```
class Barrier {
public:
    Barrier(uint max_count) : max_count_(max_count) {}

    void wait() {
        std::unique_lock<std::mutex> ul(mtx_);
        count_++;
        while (count_ < max_count_)
        {
            cv_.wait(ul);
        }
        cv_.notify_all();
    }

private:
    uint count_ = 0;
    const uint max_count_;
    std::mutex mtx_;
    std::condition_variable cv_;
};
```

In the next example, the barrier that is implemented above is used to change the `tracker`'s value. Five threads increment the `tracker` by 3 before the barrier in `IncrementByThree` function. Five other threads double the `tracker` after the barrier in `DoubleTheValue`. Without a barrier, based on how the threads are scheduled, the multiplications and additions can occur in different order leading non-deterministic results. However, by using a barrier, the calculation is synchronized and results in an expected value of 480.

```
constexpr uint num_threads = 10;
Barrier barrier(num_threads);
uint tracker = 0;
std::mutex mtx_;

void IncrementByThree() {
    std::unique_lock<std::mutex> ul(mtx_);
    tracker += 3u;
    ul.unlock();
    barrier.wait();
}

void DoubleTheValue() {
    barrier.wait();
    std::unique_lock<std::mutex> ul(mtx_);
    tracker *= 2;
}

int main() {
    std::vector<std::thread> thread_vec;

    for (int i = 0; i < 5; i++) {
        std::thread th(DoubleTheValue);
        thread_vec.push_back(std::move(th));
    }

    for (int i = 0; i < 5; i++) {
        std::thread th(IncrementByThree);
        thread_vec.push_back(std::move(th));
    }

    for (auto & th: thread_vec) {
        th.join();
    }

    printf("tracker = %d\n", tracker);

    return 0;
}
```

Note that the threads calling `DoubleTheValue()` function is deliberately instantiated before `IncrementByThree()`. But the barrier ensures that all the increments happen before the doubling.

3.5 Latch

It allows the threads to wait until some other threads complete their work. It blocks the threads calling it's `wait()` until its counter reaches zero. The counter is reduced by calling `count_down()`. A barrier releases when a certain number of threads waiting for it is reached. However, a latch releases them when its counter goes to 0.

If the counter is initialized to 1, then the thread calling `count_down()` acts as a gate keeper for the waiting thread. If the counter is initialized to N, then each of the N threads have to call `count_down()` at least once or one thread call it N times.

TODO: Add latch implementation and an example of its usage

3.6 Summary

3.6.1 Concepts

- When a thread can only start its work after a condition (that changes due to multiple threads) is met or when another thread has completed a job, a a waiting and signalling synchronization among the threads is necessary for the program to be efficient.

- A condition variable provides waiting and signalling synchronization mechanism and works along with a mutex. It is a queue for the (blocked) threads to wait. These threads continue execution upon receiving a notification from some other thread.

- A mutex and condition variable together implement a higher-level construct called monitor. In a monitor, only one thread at a time has the access to the instructions and shared resources. Since only one thread executes the instructions at a time and the shared resources are guarded by a mutex, both race conditions and data race are avoided.
- To avoid waking up of a thread due to spurious wake up calls, a predicate should be used in the condition variable.
- A mutex and condition variable can be used together to implement more synchronization mechanisms like:
  - Barrier: It lets the threads to wait until a given number of threads have come to wait and then releases all of them irrespective of the order in which the threads arrive.
  - Latch: It allows the threads to wait until its counter goes to 0.

4 Concurrency/Synchronization Bugs

4.1 Liveness

Set of properties that the concurrent programs should satisfy to make progress. A well-written program ensures that all programs will eventually make progress although they are blocked for sometime.

4.1.1 Deadlock

A member waits for another member to take an action and vice versa. Because neither members can proceed ahead a never ending non-action scenario occurs. For example, `t1` locks `mtx_a` and by the time it locks `mtx_b`, `t2` locks `mtx_b`. So each of the thread holds one lock and is waiting to lock the other mutex. Since neither of them know that their lock has to be released for the other thread to proceed ahead, it gets stuck in a never-ending, non-action phase.

```
int count = 1000;

void reduce_count(std::mutex &mtx_1, std::mutex &mtx_2) {
    while (count > 0) {
        mtx_1.lock();
        mtx_2.lock();
        if (count) {
            count--;
        }
        mtx_2.unlock();
        mtx_1.unlock();
12      }
13  }
14
15  int main() {
16      std::mutex mtx_a, mtx_b;
17      std::thread t1(reduce_count, std::ref(mtx_a), std::ref(mtx_b));
18      std::thread t2(reduce_count, std::ref(mtx_b), std::ref(mtx_a));
19      t1.join();
20      t2.join();
21      printf("Count is reduced to %d.\n", count);
22  }
```

The simplest technique to avoid deadlocks is by ensuring that the locks are taken in the same order by any thread and unlocking them in the reverse order. In the above example, line no. 18 should be modified to

```
std::thread t2(reduce_count, std::ref(mtx_a), std::ref(mtx_b));
```

With the above fix, either `t1` or `t2` locks `mtx_a` and the other one waits for it. The one which locked `mtx_a` goes ahead and locks `mtx_b` also. In a simple case like it was it was possible to lock in a predefined order, however, it may not be always feasible because a thread may not always know all the locks it needs to acquire before taking any of them.

TODO: example

4.1.1.1 `scoped_lock` To be independent of ordering, a `scoped_lock` can be used which takes the two mutexes as argument and locks them. It automatically releases them when it goes out of scope - RAII style working. The above example can be made deadlock free with as follows:

```
int count = 1000;

void reduce_count(std::mutex &mtx_1, std::mutex &mtx_2) {
    while (count > 0) {
        std::scoped_lock sc_lock(mtx_1, mtx_2);
        if (count) {
            count--;
        }
    }
}

int main() {
    std::mutex mtx_a, mtx_b;
    std::thread t1(reduce_count, std::ref(mtx_a), std::ref(mtx_b));
    std::thread t2(reduce_count, std::ref(mtx_b), std::ref(mtx_a));
    t1.join();
    t2.join();
    printf("Count is reduced to %d.\n", count);
}
```

These need not always occur. Only occurs if executions overlap and context switching from one thread to another. Conditions for deadlock:

- Mutual exclusion: a thread claims exclusive control of the resouce (mutex lock)
- Hold and wait: holds a resource and waits for another resource
- No preemption: thread cannot be made to give up its resource (cannot unlock it)
- Circular wait: there exists a cycle in the resource dependency graph

Theoretically proven that all the above four should hold for a deadlock. Knowing the above well can help avoid a deadlock.

Prevent circular wait by acquiring them in the same order in all the threads. Scoped lock wrapper sovles this problem.

4.1.2 Abandoned Lock

Just like memory resources, if it is allocated and the code ends up in a branch that it is not deallocated, then an issue occurs. The same could be possible with locks, where it is not unlocked due to the flow of code. Unlocking does not happen automatically for a mutex and hence if a thread does not unlock it, all other threads wait for ever to acquire it and end up in a never ending scenario. This can be avoided by using a wrapper around a mutex for example a scoped lock, which is the equivalent of a smart pointer. The unlocking or release of the resources happen for sure when the wrapper goes out of scope.

4.1.3 Starvation

When too many threads are competing for something, then multiple threads may never get the opportunity to access the resource even though the program successfully finishes.

4.1.4 Livelock

A livelock is similar to a deadlock in the sense that the threads block each other. However, they are different because the threads in a livelock are trying to resolve the problem but without making real progress. Livelock or deadlock can be differentiated by looking at the CPU resources the program consumes. In a deadlock, it would be close to zero but in a live lock it would be pretty high.

It can occur when two or more threads are designed to respond to the actions of each other. Both threads are doing something but their combinations of efforts does not allow to make progress and the program never reaches the end. These are often caused by algorithms that are intended to detect and recover from deadlock. TODO: Add code example. If one or more threads takes an action to avoid deadlock, then they may end up releasing some locks for the other threads and acquiring other locks. Finally, all that happens is threads acquiring and releasing locks with no real progress. To avoid it, ensure that only one thread takes action.

In C++, it can be avoided by using `yield()`, which lets the thread wait for a moment by hinting to reschedule itself and allowing other threads to run.

TODO: Need an example

4.2 Summary

4.2.1 Concepts

- A well-written concurrent program ensures that all the threads will eventually make progress even though some of them may be blocked for sometime. Following are to be avoided:
  - Deadlock: A never-ending, non-action scenario occurs because one thread waits for the other thread to take action and vice versa. As a result none of the threads proceed ahead. It can be avoided either by threads acquiring the locks in the same order or by using a `scoped_lock`.
  - Abandoned lock: The mutex is acquired but not released by a thread. Rest of the threads wait to acquire the mutex forever again leading to a never-ending, non-action scenario. It is avoidable by wrapping the mutex in a class that unlocks it when the class' object goes out of scope (RAII style).
  - Livelock: The threads are actively performing something but they don't make real progress. For example, acquiring and releasing the locks but not performing the useful computations.
  - Starvation: A thread never gets to access the resource. If the desired result can be obtained even without a thread using accessing the resource, then it is perhaps not necessary.

5 Memory model and atomic type

These convert the plain types into atomic types, which means only a single thread can read and write to its memory location.

6 Asynchronous Concurrency

6.1 Thread Pool

Upon identifying the tasks in a program that can run asynchronously, one way to possibly run them in parallel is to create independent threads or processes for each of them. Although threads are light weight, they still require some overhead in processor time and memory. In some scenarios where the task is comparably faster than creating a new thread, it can be too much of effort to create a new thread for such tasks. Rather a thread that has already completed its work can be reused to accommodate new work. This is done by a thread pool, which creates and maintains a collection of worker threads. As the program submits tasks to the thread pool, it reuses the threads that are out of work to perform new tasks and thus saving the overhead of creating and destroying new threads for every new task.

The standard library does not include thread pools at the moment. However, Boost library has it.

TODO: add example

6.2 Future

A future acts as a place holder for the result that is initially unknown but will be filled at a later point of time when the worker thread finds it out. It provides a mechanism to access the result of an asynchronous operation. The worked thread makes a promise to send the result and the parent thread holds onto a handle that provides the result.

TODO: Code example

6.2.1 Divide and Conquer

Don't spawn too many threads because of depth! Overall that does not help.

TODO: Code example with times in both ways

7 Evaluating Parallel Performance

7.1 Computational Graph

To provide an abstract representation of the program. Computational graphs help model how steps in a program relate to each other. They help visualize relationship and dependencies between the tasks. Each task is represented as a node in a graph and arrows represent the flow of execution. A task can be executed only after all nodes that feed into the task have completed their work.

The amount to which the program speeds up due to parallelization can be calculated as: take taken to execute all the tasks (work) / longest time taken though the graph (span). Minimizing span is key to designing parallel program to boost up the speed.

7.2 Speedup, latency, and throughput

7.3 Amgahl's law

7.4 Measure speedup

8 Designing Parallel Programs

A common 4-step methodology to develop a parallel solution. Can be used to design complex programs that run on large-scale parallel systems.

8.1 Partitioning

Breaking the problem down into discrete pieces of work. Simply decompose the problem into as many small problems as possible without caring about practical issues like number of processors, etc. There are two basic ways to do it:

1. Domain/data decomposition: Initially, dividing the data into small of almost equally sized partitions. Next, considering the computations to be performed and associating them with that data.

2. Functional decomposition: Considers all the computation that needs to be performed and then divides them into separate parts for different tasks to perform it. Data mapping to tasks is a secondary consideration.

Both the above compliment each other and makes sense to use a combination of them. Typically with domain decomposition because it acts as a foundation for lot of algorithms.

8.2 Communication

Determining how to coordinate execution and communicate data. If each task only needs to communicate with a small number of other tasks, point-to-point communication can be used. one acts a a producer of data and the other as consumer.

Collective communication: If a task needs to communicate with a larger group, then other procedures like broadcasting or scattering can be used. In scattering pieces of data are given to each tasks and the results are collected back. While designing such global communication systems, scaling of the system should also be considered. Because if there is one central manager for multiple worker threads and if these worker threads keep growing, then the over communication of manager with each thread slows down and can turn into a bottleneck. Creating hierarchies of workers helps - distributing the computation and communication in a way such that no single task has too much burden.

Synchronous (blocking) communication: All tasks need to wait until the entire communication is complete to continue doing other work. Can lead to waiting for a lot of time, instead of doing useful work. Asynchronous (non blocking) communication: Once a tasks sends out an asynchronous message, it can continue its work irrespective of when the receiving tasks gets the message.

Consider the following:

Overhead that the communication needs. Higher it is, lower is the availability for actual computation.

Latency is the time to send message from A to B.

Bandwidth: Data that can be communicated in unit time.

Basic multi threading, it is not of concern. However, larger programs on distributed systems, inter-system communication factors can have significant impact.

8.3 Agglomeration

Focus was on decomposing into as many tasks as possible. Allowed to consider wide range of possibilities. It is not so efficient if there are way more tasks than the processors. Group up a

8.4 Mapping

9 Summary

9.1 Checklist for Design and Implementation

- Does concurrency make sense?
  - Which part of the program limits the run time?
  - Are there parts of the program that can be executed independently or at least partially independently?
  - Can a speed up be obtained (Amgahl's law may help)?
- Does the design ensure liveness of the program? Is there a possibility of a deadlock, abandoned lock, or livelock?
- Are appropriate locks used?
- Are the best practices adheared to? If not, is there a justification?

9.2 Best Practices

condition variable wait with a predicate

9.3 Open points

Reader writer's problem

Dining philosopher's problem

Release lock before or after notification??

Which issues can be avoided using best practices, which can be caught with static analyzers, and which can't be caught using both.

References

[1] Olivia Stone and Barron Stone. *Parallel and Concurrent Programming in C++, Part 1 and Part 2*. LinkedIN Learning, 2020.

[2] Mythili Vutukuru. *PART C: Concurrency, Operating Systems*. Course Website, 2018.