C++ concurrent programming notes based off Bo's videos

Syed Paymaan Raza

November 24, 2017

Contents

1 Concurrent programming models

1. Multi-processing
    - 1 thread / process
    - Threads communicate through *Interprocess communication* channels e.g. files, types, message queues.

2. Multi-threading

    - >= 2 threads / process
    - Threads communicate through *shared memory*.
    - Pros
        - Thread faster to start (considered *light-weight process*).
        - Thread has low overhead (process needs extra protection to avoid over-stepping).
        - Communicating through shared memory faster than interprocess communication.
        - Overall, multi-threading provides better performance than multi-processing.
    - Cons
        - Difficult to implement (have to deal with thread specific issues).
        - Can't run on distributed systems; multi-processing on the other hand can be easily distributed on such systems and therefore, run concurrently.
            * Main reason for this is that shared memory blocks concurrent distribution

In practice, within the same program, one can expect to see a mixture of both models i.e. some processes are single-threaded (Multi-processing model) and others are multi-threaded (Multi-threading model).

In these notes, we'll mainly talk about the Multi-threading model.

2 Getting started example

```
#include <iostream>
#include <thread>

void f1() {
    std::cout << "message" << std::endl;
}
int main() {
    f1();
    return 0;
}
```

```
message
```

This program is not multi-threaded (only 1 thread running). Let's make this program multi-threaded:

```
#include <iostream>
#include <thread>

void f1() {
    std::cout << "message" << std::endl;
}
int main() {
    std::thread t1(f1); // t1 thread (child of main thread) starts running
    t1.join();          // main thread waits for t1 to finish
    return 0;
}
```

```
message
```

Suppose t1 is long running and main thread doesn't want to wait:

```
#include <iostream>
#include <thread>

void f1() {
    std::cout << "message" << std::endl;
}
int main() {
    std::thread t1(f1); // t1 thread (child of main thread) starts running
    t1.detach();        // t1 runs freely on its own (daemon process).
    return 0;
}
```

Before getting to the result, note that detatch causes a daemon process to be created. This means that when t1 is finished, since it's not connected to the main thread, the C++ runtime library will reclaim the resource. Note however that sometimes daemon process keep running until the machine is shut down.

Back to the result. We can see the nothing is printed. This is because the main thread ran so quickly that it finished before t1 could print its message.

To avoid this un-determinism between two independent threads, we can add synchronization (later).

Some points:

- main() and main thread *owns* f1() and t1 thread.
- calling detach however => main thread and t1 thread independent despite of owner or parent-child relationship
- typically, the owner should outlive its children.
- we can join and detach threads only once
    - => can not call join after detach; once detached, always detached.
    - can check if joinable though: t1.joibable

3 Thread management

Our last example was:

```
#include <iostream>
#include <thread>

void f1() {
    std::cout << "message" << std::endl;
}
int main() {
    std::thread t1(f1);
    t1.detach();        // or t1.join()
    return 0;
}
```

Note that we have to make a decision to either join or detach the thread after its created (and hence starts running). The decision has to be made before the thread object goes out of scope.

Let's look at an example:

```
#include <iostream>
#include <thread>

void f1() {
    std::cout << "f1" << std::endl;
}
int main() {
    std::thread t1(f1);
    // main thread  work while t1 is running
    try {
        for (int i = 0; i < 10; ++i)
            std::cout << "main: " << i << std::endl;
    } catch (...) {
        t1.join();
        throw; // rethrow the exception: hopefully someone else will catch and
               // handle it
    }
    t1.join(); // wait for t1 to finish
    return 0;
}
```

```
main:  0
main:  1
main:  2
main:  3
main:  4
main:  5
main:  6
main:  7
main:  8
main:  9
f1
```

Note that:

- we had to use try/catch for main thread's work because if we don't do that and then if the work throws an exception, t1 will go out of scope before being joined or detached.
- an alternative of try/catch here to ensure t1 is joined is wrapping the work in a class and use RAII

We saw that threads can be instanitated and hence associated with functions. In general, threads can be associated with any callable object. Let's take a look at an example where the callable object is a Functor class:

```cpp
#include <iostream>
#include <string>
#include <thread>

class Fctor {
  public:
    void operator()(const std::string& msg) {
        std::cout << "t1: " << msg << std::endl;
    }
};

int main() {
    const std::string s = "Answer to life is 42";
    std::thread t1((Fctor()), s);
    try {
        for (int i = 0; i < 10; ++i)
            std::cout << "main: " << s << std::endl;
    } catch (...) {
        t1.join();
        throw;
    }
    t1.join();
    return 0;
}
```

```
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
main: Answer to life is 42
t1:   Answer to life is 42
```

Note that even though Fctor argument is pass-by-reference, the parameter is passed by value. This is because parameter to the thread is always passed by value. If passing by reference is really needed, use std::ref to wrap the callsite argument. Another option is to pass a pointer.

Also note that ideally, threads should share minimum memory to avoid data races. So in the earlier example, if s is not used in the main thread, we can use std::move at the thread creation callsite to change the ownership of s from the main thread to t1. This is both safe and efficient. In C++, there are objects that can not be copied but can be moved. An example is is the thread object itself i.e. std::thread t2 = t1 won't work but std::thread t2 = std::move(t1) will; it will move the ownership of t1 to t2; t1 would then become empty.

Each thread also has a unique (?) identification number associated with it. To get that numer, we can use std::this$_{\text{thread}}$::get$_{\text{id}}$() which will print current thread's id. To print a specific thread's id, we can use t1.get$_{\text{id}}$().

4 Data races and Mutex

Let's take an example:

```
#include <iostream>
#include <thread>

void f1() {
    for (int i = 0; i > -10; --i)
        std::cout << "f1: " << i << std::endl;
}
int main() {
    std::thread t1(f1);
    for (int i = 0; i < 10; ++i)
        std::cout << "main: " << i << std::endl;
    t1.join();
    return 0;
}
```

```
main:   0
main:   1
main:   2
main:   3
main:   4
main:   5
main:   6
main:   7
main:   8
main:   9
f1:     0
f1:    -1
f1:    -2
f1:    -3
f1:    -4
f1:    -5
f1:    -6
f1:    -7
f1:    -8
f1:    -9
```

```
main: 0 f1: main: 1 0 f1: -1 f1: -2 f1: -3 f1: main: -42
    f1: main: -53
    f1: main: -64
    f1: main: -75
    f1: main: -86
    f1: main: -97
    main: 8 main: 9
```

The reason we get this garbled output is that there are two threads running and writing to cout (std output) at the same time. In other words, both threads are racing for a common resource, cout. This results in a race condition which means the outcome of the program depends on the relative execution order of one or more threads. This by defintion is un-determinstic.

One way to solve to race condition is to use mutex which synchronizes the access of the common resource:

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>
```

```
std::mutex mu;
void shared_print(const std::string& msg, const int id) {
    mu.lock();
    std::cout << msg << id << std::endl;
    mu.unlock();
}
void f1() {
    for (int i = 0; i > -10; --i)
        shared_print("f1: ", i);
}
int main() {
    std::thread t1(f1);
    for (int i = 0; i < 10; ++i)
        shared_print("main: ", i);
    t1.join();
    return 0;
}
```

```
main:   0
main:   1
main:   2
main:   3
main:   4
main:   5
main:   6
main:   7
main:   8
main:   9
f1:     0
f1:    -1
f1:    -2
f1:    -3
f1:    -4
f1:    -5
f1:    -6
f1:    -7
f1:    -8
f1:    -9
```

main: 0 main: 1 f1: 0 main: 2 f1: -1 main: 3 f1: -2 main: 4 f1: -3 main: 5

f1: -4 main: 6 f1: -5 main: 7 f1: -6 main: 8 f1: -7 main: 9 f1: -8 f1: -9

Now we can see that only both threads queue up and wait for each other before executing. This is achieved using lock and unlock mechanism of the $\text{shared}_{\text{print}}$ resource.

There is a problem with the above code though. If the $\text{shared}_{\text{print}}$ cout code throws an exception, the mutex will remain locked throughout the program. To fix this issue:

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

std::mutex mu;
void shared_print(const std::string& msg, const int id) {
    std::lock_guard<std::mutex> guard(mu); // RAII
    std::cout << msg << id << std::endl;
}
void f1() {
    for (int i = 0; i > -10; --i)
        shared_print("f1: ", i);
}
int main() {
    std::thread t1(f1);
    for (int i = 0; i < 10; ++i)
        shared_print("main: ", i);
    t1.join();
    return 0;
}
```

```
main:   0
main:   1
main:   2
main:   3
main:   4
main:   5
main:   6
main:   7
main:   8
main:   9
f1:     0
f1:    -1
f1:    -2
f1:    -3
f1:    -4
f1:    -5
f1:    -6
f1:    -7
f1:    -8
f1:    -9
```

main: 0 f1: 0 main: 1 f1: -1 main: 2 f1: -2 main: 3 f1: -3 main: 4 f1: -4 main: 5 f1: -5 main: 6 f1: -6 main: 7 f1: -7 main: 8 f1: -8 main: 9 f1: -9

Here, RAII implies that once guard is destructed or goes out of scope, the destructor automatically unlocks the mutex, mu.

Another problem with this example is that since cout is a global variable/resource, someone else can access cout without going through $\text{shared}_{\text{print}}$.

Although cout is a global stream and it's hard to fully bind it to a mutex, other things can be bounded:

```
class LogFile {
  public:
    LogFile() {
        f.open("log.txt");
    }
    ~LogFile() {
        f.close();
    }
    void shared_print(const std::string& msg, const int id) {
        std::lock_guard<std::mutex> locker(m_mutex);
        f << "From " << msg << ": " << id << std::endl;
    }

  private:
    std::mutex m_mutex;
    std::ofstream f;
};

void f1(LogFile& log) {
    for (int i = 0; i < 100; ++i)
        log.shared_print("f1: ", i);
}

int main() {
    LogFile log;
    std::thread t1(f1, std::ref(log));
    return 0;
}
```

Now, we can only access the resource `f` via mutex. Note that it's a bad idea to expose this resource e.g. using a getter since the clients can then use it without going through the mutex.

Now let's assume that we have avoided leaking the resource by abstracting in a class, does it guarentee that our program is thread-safe i.e. there is no race condition?

Let's look at a STL example:

```
class Stack {
public:
  void pop();
  int top();
private:
  int* _data;
  std::mutex _mu;
};

void f1(Stack& st) {
  int v = st.top();
  st.pop();
  process(v);
}
```

Assume that pop() and top() access _data through the mutex. This code is not thread-safe even though we have protechted our resource ($_{\text{data}}$) using a mutex. The reason is that 2 threads can call f1, which calls st.pop() and get the same stack value. The reason is that although we have used mutex to synchronize data access, the interface is inherently not thread-safe i.e. top() will return the same value if called twice. One possible solution is to combine top() and pop() athlought it then breaks the "one function should do one thing only" principle.

Note that although combining the two functions to something like int pop() would make the program thread safe, it would still not be exception safe because if one thread calls pop() and there is an exception thrown, the lock will remain locked until the end of the program. This is why C++ STL doesn't return a value in std::stack pop()'s implementation.

5 Deadlock

Mutex is a lock which provieds locking mechanism to threads. Now we have 2 mutexes as well. That means that the resource can be accessed only when both mutexes are in an unlocked state (note that locked and unlocked are the only two states for mutexes).

However, using more than one mutex can sometimes lead to *Deadlock*:

```
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

std::mutex mu;
std::mutex mu2;
void shared_print(const std::string& msg, const int id) {
    std::lock_guard<std::mutex> guard(mu); // RAII
    std::lock_guard<std::mutex> guard2(mu2); // RAII
    std::cout << msg << id << std::endl;
}
void shared_print2(const std::string& msg, const int id) {
    std::lock_guard<std::mutex> guard2(mu2); // RAII
    std::lock_guard<std::mutex> guard(mu); // RAII
```

```cpp
    std::cout << msg << id << std::endl;
}
void f1() {
    for (int i = 0; i > -100; --i)
        shared_print2("f1: ", i);
}
int main() {
    std::thread t1(f1);
    for (int i = 0; i < 100; ++i)
        shared_print("main: ", i);
    t1.join();
    return 0;
}
```

```
main:    0
main:    1
main:    2
main:    3
main:    4
main:    5
main:    6
main:    7
main:    8
main:    9
main:   10
main:   11
main:   12
main:   13
main:   14
main:   15
main:   16
main:   17
main:   18
main:   19
main:   20
main:   21
main:   22
main:   23
main:   24
main:   25
main:   26
main:   27
main:   28
main:   29
main:   30
main:   31
main:   32
main:   33
main:   34
main:   35
main:   36
main:   37
main:   38
main:   39
main:   40
main:   41
main:   42
main:   43
main:   44
main:   45
main:   46
main:   47
main:   48
```

Notice that the program got stuck while printing and we had to C-c to terminate the program. This happened because in `shared_print`, we locked `mu` and then `mu2` and vice versa in `shared_print2`. Since both of the functions are associated with threads that are running at the same time, this means that there was an instance e.g. `shared_print` locked `mu` but before locking `mu2`, `shared_print2` locked it. Now `shared_print` has to wait before `shared_print2` unlock it but `shared_print2` itself locked `mu2` and `shared_print` locked `mu` before it could lock `mu` so `shared_print2` is also waiting for `shared_print`. Now both functions are waiting for each other and therefore, we are in a deadlock situtation.

One possible solution is to use the same order of mutex locking in both functions.

C++ standard library has provided a better solution `std::lock` which can lock arbitrary number of mutexes with deadlock avoiding mechanisms on top:

```
std::lock(mu, mu2);
std::lock_guard<std::mutex> locker(mu, std::adopt_lock);
std::lock_guard<std::mutex> locker2(mu2, std::adopt_lock);
```

`std::adopt_lock` tells the locker that the mutex is already locked and all you (locker) needs to do is to adopt the ownership of the mutex, so that when you go out of scope, remember to unlock the mutex.

Other solutions to avoid deadlocks:

- Consider if you really need two lockers at the same time, else prefer locking single mutex at a time:

```
{
std::lock_guard<std::mutex> locker(mu);
// do work
}
{
std::lock_guard<std::mutex> locker2(mu2);
// do work
}
```

- Avoid locking a mutex and then calling a user provided function

Lock granularity:

- Fine-grained lock: protects small amount of data
- Coarse-grained lock: protects large amount of data

6 Unique$_{\text{lock}}$ and lazy initialization

We can use `unique_lock` instead of `lock_guard` as follows:

```
void shared_print(const std::string& id, const int val) {
    // std::lock_guard<std::mutex> locker(mu);
    // std::unique_lock<std::mutex> locker(mu);
    std::unique_lock<std::mutex> locker(mu, std::defer_lock);

    // do something else

    locker.lock();
    // use resource (which needed lock protection)
    locker.unlock();

    // lock again
    locker.lock();

    // can move but not copy
    std::unique_lock<std::mutex> locker2 = std::move(lock);

    // rest of the code
}
```

As we can see, `unique_lock` is more flexible in terms of when we can lock and unlock. It can also allow multiple locks and unlocks. The downside of using it over `lock_guard` is performance since it's more heavy weight.

Let's look at another example using lazy initialization:

```
void shared_print(const std::string& id, const int val) {
    if (!f.is_open()) {
        f.open("log.txt"); // only open file once
                           // lazy initialization
                           // initialization upon first use idiom
    }
    std::unique_lock<std::mutex> locker(mu);
    f << "some string" << std::endl;
    locker.unlock();
}
```

Here, we are protecting by locking the printing to `f` but opening `f` is not protected so multiple threads can open the file at the same time which is undesirable. To fix this issue, we can move the `locker` up so that it protects opening `f` too but that's not right since we open the file once while printing is done everytime the function calls.

So one solution maybe to use another mutex `mu_open`:

```
void shared_print(const std::string& id, const int val) {
    if (!f.is_open()) {
        std::unqiue_lock<std::mutex> locker2(mu_open);
        f.open("log.txt");
    }
    std::unique_lock<std::mutex> locker(mu);
    f << "some string" << std::endl;
    locker.unlock();
}
```

This program is still not thread safe since since `!f.is_open()` is not protected. Let's do that:

```
void shared_print(const std::string& id, const int val) {
    {
        std::unqiue_lock<std::mutex> locker2(mu_open);
        if (!f.is_open()) {
            f.open("log.txt");
        }
    }
    std::unique_lock<std::mutex> locker(mu);
    f << "some string" << std::endl;
    locker.unlock();
}
```

This program is now thread-safe but inefficient since every thread will do the locking and then checking if file is open. C++ provides a better way in `std::once_flag flag` which would also eliminate the need for an extra mutex for one-time checking:

```
void shared_print(const std::string& id, const int val) {
    // file will be opened once by one (first) thread
    std::call_once(flag, [&](){f.open("log.txt");});

    std::unique_lock<std::mutex> locker(mu);
    f << "some string" << std::endl;
    locker.unlock();
}
```

7 Condition variables

Let's look at this example:

```
#include <chrono>
#include <deque>
#include <iostream>
#include <mutex>
#include <thread>

std::deque<int> q;
std::mutex mu;

void f1() {
    int count = 10;
    while (count > 0) {
        std::unique_lock<std::mutex> locker(mu);
        q.push_front(count);
        locker.unlock();
        std::this_thread::sleep_for(std::chrono::seconds(1));
        count--;
    }
}

void f2() {
    int data = 0;
    while (data != 1) {
        std::unique_lock<std::mutex> locker(mu);
        if (!q.empty()) {
            data = q.back();
            q.pop_back();
            locker.unlock();
            std::cout << "f2 got a value from f1: " << data << std::endl;
        } else {
            locker.unlock();
        }
    }
}

int main() {
    std::thread t1(f1);
    std::thread t2(f2);
    t1.join();
    t2.join();
    return 0;
}
```

```
f2 got a value from f1: 10
f2 got a value from f1:  9
f2 got a value from f1:  8
f2 got a value from f1:  7
f2 got a value from f1:  6
f2 got a value from f1:  5
f2 got a value from f1:  4
f2 got a value from f1:  3
f2 got a value from f1:  2
f2 got a value from f1:  1
```

As we can see, we have two threads here and `f2` is a *consumer* which gets queue values from the *producer*, `f1`. Also, the resource `q` is shared between the two threads so it's a good that we are using locking synchronization using `unique_lock<mutex>`.

However, there is an issue: `thread2` (corresponding to `f2`) is in a busy waiting state since it's keep checking if `q` is empty; if `q` is empty, it will unlock the locker and *immediately* go to the next loop; this busy waiting cycles are very inefficient.

To make the program more efficient, one way is:

```
    std::cout << "f2 got a value from f1: " << data << std::endl;
} else {
    locker.unlock();
    std::this_thread::sleep_for(std::chrono::milliseconds(10));
}
```

The challenge here is picking the time constant e.g. `10` in this example. If the time is too short, then the thread will still end up time looping; if the time is too long, then the thread will not be able to get the data in time.

Another elegant way to solve this issue is using a *condition variable*, which is another way of synchronization apart from mutexes. Here's how the code changes:

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

std::deque<int> q;
std::mutex mu;
std::condition_variable cond;

void f1() {
    int count = 10;
    while (count > 0) {
        std::unique_lock<std::mutex> locker(mu);
        q.push_front(count);
        locker.unlock();
        cond.notify_one(); // notify one waiting thread, if any
        std::this_thread::sleep_for(std::chrono::seconds(1));
        count--;
    }
}

void f2() {
    int data = 0;
    while (data != 1) {
        std::unique_lock<std::mutex> locker(mu);
        cond.wait(locker); // put thread into sleep until notified by cond in
                           // thread1 (f1)
        data = q.back();
        q.pop_back();
        locker.unlock();
        std::cout << "f2 got a value from f1: " << data << std::endl;
    }
}
```

```
int main() {
    std::thread t1(f1);
    std::thread t2(f2);
    t1.join();
    t2.join();
    return 0;
}
```

```
f2 got a value from f1: 10
f2 got a value from f1:  9
f2 got a value from f1:  8
f2 got a value from f1:  7
f2 got a value from f1:  6
f2 got a value from f1:  5
f2 got a value from f1:  4
f2 got a value from f1:  3
f2 got a value from f1:  2
f2 got a value from f1:  1
```

In essence, condition variables can enforce that thread2 will go ahead and fetch the data only when thread1 has pushed the data into the queue. In other words, it can be enforce certain parts of the two threads to be exectured in a pre-defined order.

Another question is that why we need to pass the `locker` to `cond.wait`. The answer is that doing so automatically locks and unlocks the mutex before and after waiting respectively. Not doing so would mean that we can lock out other threads while we ourselves our sleeping, which is not desirable here. This also implies that we must use `unique_lock` since we'll be locking and unlocking multiple times.

Things look good as long as thread2, while sleeping, can only be waked up by condition variable in thread1. However, that's not totally true since thread2 can wake by itself, which is called *spurious wake*. We need to make sure that this doesn't happen and that thread2 goes to sleep again unless it was condition variable that woke it up, or equivalently, if `q` is not empty:

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

```cpp
std::deque<int> q;
std::mutex mu;
std::condition_variable cond;

void f1() {
    int count = 10;
    while (count > 0) {
        std::unique_lock<std::mutex> locker(mu);
        q.push_front(count);
        locker.unlock();
        cond.notify_one(); // notify one waiting thread, if any
        std::this_thread::sleep_for(std::chrono::seconds(1));
        count--;
    }
}

void f2() {
    int data = 0;
    while (data != 1) {
        std::unique_lock<std::mutex> locker(mu);
        cond.wait(locker, [](){return !q.empty();}); // spurious wake
        data = q.back();
        q.pop_back();
        locker.unlock();
        std::cout << "f2 got a value from f1: " << data << std::endl;
    }
}

int main() {
    std::thread t1(f1);
    std::thread t2(f2);
    t1.join();
    t2.join();
    return 0;
}
```

```
f2 got a value from f1: 10
f2 got a value from f1:  9
f2 got a value from f1:  8
f2 got a value from f1:  7
f2 got a value from f1:  6
f2 got a value from f1:  5
f2 got a value from f1:  4
f2 got a value from f1:  3
f2 got a value from f1:  2
f2 got a value from f1:  1
```

Note that we now passed an additional predicate to `cond.wait`.

Another thing to note is that if multiple threads are waiting on `cond` here, then `cond` will only wake up *one* thread since we're using `cond.notify_one()`. If we want *all* the threads to be awake, use `cond.notify_all()`.

In summary, condition variables are used to synchronize the execution sequence of threads.ooo

8 Future and Promise

Let's look at a simple factorial program with two threads:

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

void fact(int n) {
    int res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "fact(" << n << ") = " << res << std::endl;
}

int main() {
    std::thread t1(fact, 4);
    t1.join();
    return 0;
}
```

```
fact(4) = 24
```

Here, t1 is computing the factorial of 4. But let's say we want to return the result of factorial from the child thread to the parent thread. One way is:

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

void fact(int n, int& x) {
    int res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "fact(" << n << ") = " << res << std::endl;
    x = res;
}

int main() {
    int x;
    std::thread t1(fact, 4, std::ref(x));
    t1.join();
    return 0;
}
```

```
fact(4) = 24
```

This is not enough though. `x` is a shared resource between main and child threads, so we need to lock its access using a mutex. We also want to ensure using a condition variable that the child thread computes the result *first* and only then the main thread fetches it. Here's how the code may look like:

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

std::mutex mu;
std::condition_variable cond;
```

```
void fact(int n, int& x) {
    int res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "fact(" << n << ") = " << res << std::endl;
    x = res;
}

int main() {
    int x;
    std::thread t1(fact, 4, std::ref(x));
    t1.join();
    return 0;
}
```

```
fact(4) = 24
```

The code isn't complete yet: we have to lock/unlock the mutex, use the condition variable for notification. It's also not that good of a design because of 2 global variables. A neater and more elegant approach is using the C++ standard library's `std::async` function, `std::future`, `std::promise`:

```
#include <future>
#include <iostream>
#include <mutex>
#include <thread>

int fact(int n) {
    int res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "child thread: fact(" << n << ") = " << res << std::endl;
    return res;
}

int main() {
    int x;
    std::future<int> fu = std::async(std::launch::async, fact, 4);
    x                   = fu.get();
    std::cout << "main thread: fact(" << "4" << ") = " << x << std::endl;
    return 0;
}
```

```
child  thread:  fact(4)  =  24
main   thread:  fact(4)  =  24
```

`std::async` returns a `std::future` object which represents "getting something from the future". `std::launch::async` means *always* create a thread; alternatively, we can use `std::launch::deferred` to *never* create a thread; or use `std::launch::async | std::launch::deferred` to decide creation of thread based on implementatiom. `fu.get()` will wait until child thread finishes and only then return the returned value from the child thread to the main thread. Note that we can only use `fu.get` once; doing so more would result in a crash.

We can also use `std::future` to do the opposite: pass the value from the main thread to the child thread; not at the time of creating the thread, but sometime in the *future*; for that we also need `std::promise`. Here's how the code looks like:

```
#include <future>
#include <iostream>
#include <mutex>
#include <thread>

int fact(std::future<int>& f) {
    int res = 1;
    int n = f.get();
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "child thread: fact(" << n << ") = " << res << std::endl;
    return res;
}

int main() {
    int x;
    std::promise<int> p;
    std::future<int> f  = p.get_future();
    std::future<int> fu = std::async(std::launch::async, fact, std::ref(f));

    // do something else
    p.set_value(4);
    x = fu.get();
    std::cout << "main thread: fact("
              << "4"
              << ") = " << x << std::endl;
    return 0;
}
```

```
child thread: fact(4) = 24
main thread: fact(4) = 24
```

If we *break* the promise i.e. don't do `p.set_value(4)`, we'll get a runtime exception `std::future_errc:broken_promise`. If we really don't have a value to provide, then the best we can do is specify a runtime exception to be thrown using `p.set_exception(std::make_exception_ptr(std::runtime_error("no value, had to break promise :(")));`.

Note that just like `thread` and `unique_lock`, `future` and `promise` can only be moved, not copied.

Now suppose we want to compute factorial multiple times:

```
#include <future>
#include <iostream>
#include <mutex>
#include <thread>

int fact(std::future<int>& f) {
    int res = 1;
    int n   = f.get();
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "child thread: fact(" << n << ") = " << res << std::endl;
    return res;
}

int main() {
    int x;
    std::promise<int> p;
    std::future<int> f   = p.get_future();
    std::future<int> fu  = std::async(std::launch::async, fact, std::ref(f));
    std::future<int> fu2 = std::async(std::launch::async, fact, std::ref(f));
    std::future<int> fu3 = std::async(std::launch::async, fact, std::ref(f));
    // ... 10 threads

    // do something else

    p.set_value(4);
    x = fu.get();
    std::cout << "main thread: fact("
              << "4"
              << ") = " << x << std::endl;
    return 0;
}
```

This won't work because each future can call the `get` function only *once*.

One solution is to create 10 promises and 10 futures and have fu, fu2, fu3 etc. This is not ideal because it's not scalable.

A better solution provided by C++ standard library is `std::shared_future` which unlike normal `std::future`, can be copied, so it can be passed by value:

```
#include <future>
#include <iostream>
#include <mutex>
#include <thread>

int fact(std::shared_future<int> f) {
    int res = 1;
    int n   = f.get();
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "child thread: fact(" << n << ") = " << res << std::endl;
    return res;
}

int main() {
    int x;
    std::promise<int> p;
    std::future<int> f         = p.get_future();
    std::shared_future<int> sf = f.share();
    std::future<int> fu  = std::async(std::launch::async, fact, sf);
    std::future<int> fu2 = std::async(std::launch::async, fact, sf);
    std::future<int> fu3 = std::async(std::launch::async, fact, sf);
    // ... 10 threads

    // do something else

    p.set_value(4);
    x = fu.get();
    std::cout << "main thread: fact("
              << "4"
              << ") = " << x << std::endl;
    return 0;
}
```

```
child   thread:   fact(4)   =   24
child   thread:   fact(4)   =   24
child   thread:   fact(4)   =   24
main    thread:   fact(4)   =   24
```

Now, when the main/parent sets the value 4 using `set.value()`, all the child threads get the same value when they call the `get` function. So, `shared_future` is very handy when it comes to broadcast type of model.

9 Callable objects

C++ defines a lot of things as callable. Here's a summary of the implications of using various callable objects with respect to threads:

```
class A {
  public:
    void f(int x, char c) {}
    long g(double x) {
        return 0;
    }
    int operator()(int N) {
        return N * N;
    }
};

void foo(int x) {}
```

```
int main() {
  A a; // Note A is a functor class because of operator()()
  std::thread t1(a, 6); // copy_of_a() in a different thread
  std::thread t2(std::ref(a), 6); // a() in a different thread
  std::thread t3(std::move(a), 6); // same as above + a is no longer usable in main th
  std::thread t4(A(), 6); // creates temp A and then copy_of_temp_a() in a different t
  std::thread t5([](int x){return x*x}); // runs lambda function in a different thread
  std::thread t6(foo, 7); // runs foo function in a different thread
  std::thread t7(&A::f, a, 8, 'w'); // copy_of_a.f(8, 'w') in a different thread
  std::thread t8(&A::f, &a, 8, 'w'); // a.f(8, 'w') in a different thread

  // other similar things
  // std::bind(a, 6);
  // std::async, std::call_once etc.

  return 0;
}
```

10 packaged$_{\text{task}}$

`packaged_task` is another way to get future like `async` and `promise`. For more details on tradeoffs, see these links:

1. When to use promise over async or packaged$_{\text{task}}$?
2. What is the difference between packaged$_{\text{task}}$ and async?
3. Difference between promise, packaged task and async
4. Packaged task

Here's an example on how to use `packaged_task` in a threading context by associating the task with a function object using `bind`:

```
#include <deque>
#include <functional>
#include <future>
#include <iostream>
#include <mutex>
#include <thread>
```

```cpp
int fact(int n) {
    int res = 1;
    for (int i = 2; i <= n; ++i)
        res *= i;
    std::cout << "child thread: fact(" << n << ") = " << res << std::endl;
    return res;
}

std::deque<std::packaged_task<int()>> task_q;
std::mutex mu;
std::condition_variable cond;

void f1() {
    std::packaged_task<int()> t;
    {
        std::unique_lock<std::mutex> locker(mu);
        cond.wait(locker, []() { return !task_q.empty(); });
        // note that front and pop have to be under the same locker
        // to be thread safe
        t = std::move(task_q.front());
        task_q.pop_front();
    }
    t();
}

int main() {
    std::thread t1(f1);
    std::packaged_task<int()> t(std::bind(fact, 4));
    std::future<int> fu = t.get_future();
    {
        std::lock_guard<std::mutex> locker(mu);
        task_q.push_back(std::move(t));
    }
    cond.notify_one();
    int x = fu.get();
    std::cout << "main thread: fact(4) = " << x << std::endl;
    t1.join();
    return 0;
}
```

```
child   thread:   fact(4)   =   24
main    thread:   fact(4)   =   24
```

So, in summary, there are 3 main ways to get future:

1. promise: get$_{\text{future}}$()
2. packaged$_{\text{task}}$: get$_{\text{future}}$()
3. async() returns a future