Conception Objets Avancée

C++ threads

Giuseppe Lipari

CRIStAL - Université de Lille

Outline

Threads

Mutual exclusion

Condition variables

Asynchronous functions, future and promises

Performance issues

Outline

Creating a thread

- A thread is just an object of class `std::thread`
- To create a thread, you have to pass a callable (a function or a function object) to the constructor

```
void body() { ... }
void g()
{
    std::thread some_thread(body);
    ...
    // wait for the thread to complete
    some_thread.join();
}
```

Creating a thread, II

- The callable is copied into the thread object, so it is safe to destroy it afterwards

```
struct callable {
    void operator()() { ... }
};

std::thread ok_function()
{
    callable x;
    return std::thread(x);
} // x is destroyed, but it has been copied, so this is ok
```

Arguments

- Arguments are copied too, so (unless they are references), it is ok to destroy them afterwards

```
class A {...};

void myfunction(A &param) { /* thread body */ }

std::thread f()
{
    A obj;
    return std::thread(myfunction, obj);
}
```

- see examples/thread_object_end.cpp

Examples

- Creating multiple threads
  - See `examples/threadex.cpp`
- A functional way to start threads
  - See `examples/thread_ret.cpp`
- A class that implements an active object
  - See `examples/thread_class.cpp`

Copying threads

- Threads objects cannot be copied
  - However, they support the move constructor
  - In other words, threads are never copied (with the result of having two threads), but actually moved
  - this allows thread objects to be returned from functions without caring about ownership

```
std::thread make_thread()
{
     return std::thread(fun, arg);
}

int main() {
    std::thread th = make_thread();
    // there is only one thread active at this point
}
```

Exceptions in threads

- If a callable passed to `std::thread` throws an exception, and the exception is not caught, the entire program is terminated
- See `examples/thread_exception.cpp`

Destroying the thread object

- What happens if the object is destroyed before the thread has terminated?
  - see examples/thread_nodetach.cpp

```
void body()
{
    cout << "This is a thread" << endl;
    this_thread::sleep_for( chrono::seconds(1) );
    cout << "Am I still alive? " << endl;
}
void make_thread()
{
    thread mythread(body);
}
int main()
{
    make_thread();
    this_thread::sleep_for( chrono::seconds(3) );
    return 0;
}
```

Detaching vs. joining

- In the previous example,
  - the abort is triggered by the fact that every thread must be joined by default
  - if we lose the reference to the thread object, we cannot join the thread anymore
- If this is the behaviour we need, then we have to detach the thread
  - this means that the thread continues execution, but it is not possible to join it anymore
  - it is possible to explicitly detach a thread by calling `detach` on the object.
  - in that case, the object becomes *not a thread*
  - if you do a join on a *not a thread* object, the join returns immediately
    - see how we can modify the above function to detach the thread

Thread local

- Thread local variables are special variables that are allocated when a thread is created, and deleted when the thread terminates
  - There is exactly one copy per each thread

```
thread_local int index = 0;
void body(int x)
{
    index = x;
    cout << "My index is " << index << endl;
    ...
}
void fun()
{
    // the value printed depends on which
    // thread calls this function
    cout << index << endl;
}
```

Outline

Mutexes

- C++ supports different kind of mutexes
  - simple, recursive, shared
  - each one can be timed or not
- Mutexes can be locked directly by using one of the following
  - `lock`, `try_lock`, `unlock`
  - A timed mutex also support `try_lock_for` and `try_lock_until` for a time-out
  - A shared mutex also supports `lock_shared` and `try_lock_shared`

Simple mutex

- A mutex is used to avoid the possibility that several threads execute the same code
  - typically, this is sensible code that should not be executed at the same time by different threads, otherwise some inconsistency may arise
- An "stupid" example in the previous code(`examples/thread_local.cpp`): we do not want to mix the output of the different threads

Guards

- Instead of accessing mutexes directly, it is useful to use guards for the RAII technique
  - the simplest one is `lock_guard`

```
std::mutex m;

void function()
{
    std::lock_guard<std::mutex> l(m);
    // critical section of code
    // if an exception is thrown,
    //  the mutex is automatically unlocked
}
```

Unique lock

- `unique_lock` allows also to directly release and acquire the ownership several times
  - when destroyed, it will unlock if necessary

```
std::mutex m;

void function()
{
    std::lock_unique<std::mutex> l(m);
    // critical section of code
    l.unlock();
    // normal code;
    l.lock();
    // critical section again
}
```

Recursive mutex

- A recursive mutex can be useful if we need to lock the same mutex several times from the thread
  - for example in a recursive function

```
int recursive_function(int x)
{
    if (x == 0) {
        std::unique_lock<std::recursive_mutex> l(m);
    std::cout << "End of recursion" << std::endl;
    return x;
    }
    else {
        std::unique_lock<std::recursive_mutex> l(m);
    cout << "This is recursive_function(" << x << ");" << endl;
    global++;
    return recursive_function(--x);
    }
}
```

Another example

- in the following example, function `f()` can be called from the `main()` or from function `g()`

```
class Shared {
private:
    std::recursive_mutex m;
public:
    Shared() {}
    Shared(const Shared &) = delete;
    Shared & operator=(const Shared &) = delete;

    void f() {
        std::unique_lock<std::recursive_mutex> l(m);
        std::cout << "Inside function f() " << std::endl;
    }
    void g() {
        std::unique_lock<std::recursive_mutex> l(m);
        std::cout << "Inside function g()" << std::endl;
        f();
    }
};
```

Direct locking

- It is possible to lock/unlock mutexes using global functions
- This can be used when we have to lock several mutexes at once

```
struct bank_account {
    explicit bank_account(int balance) : balance(balance) {}
    int balance;
    std::mutex m;
};

void transfer(bank_account &from, bank_account &to, int amount)
{
    // attempts to lock both mutexes without deadlock
    std::lock(from.m, to.m);
    // code
    std::unlock(from.m, to.m);
}
```

Executing code only once

- It is possible to specify that a certain function must be called only once

```
std::once_flag flag;

void do_once()
{
    std::call_once(flag, [](){ std::cout << "Called once" << std::endl; });
}

int main()
{
    std::thread t1(do_once);
    std::thread t2(do_once);
    std::thread t3(do_once);

    t1.join();
    t2.join();
    t3.join();
}
```

Shared mutex

- A shared mutex allows to implement the reader/writer paradigm
- Suppose you have several threads that want to read from a data structure and a few writers that want to write on the data structure
  - we must avoid that two writers, or a reader and a writer access the data structure at the same time
  - however, readers can actually read from the data structure concurrently

	Reader	Writer
Reader	Ok	NO
Writer	NO	NO

Shared mutex - II

- Using simple mutexes
  - every thread is blocked if a thread is access a data structure
  - this means that two readers cannot read at the same time
  - This is a safe solution, but it is not very efficient
- Shared mutex
  - writers use the normal `lock` operations, while readers use the special `shared_lock` operation
  - a `shared_lock` operation is blocking only if someone has already done a normal `lock` operation
- Only available from C++17

Outline

Conditions

- Condition variables are used to synchronize threads
  - They must always be paired with one mutex
- Interface:
  - default constructor only, cannot be copied or moved
  - `wait`: takes a `unique_lock` as parameter, and unlocks it while the thread is blocked

```cpp
std::mutex m;
std::condition_variable cv;

int fun()
{
    std::unique_lock l(m);
    while (condition) cv.wait(l);

    // can execute critical section code
}
```

Interface

- Interface (cont)
  - `wait_for`: a timed version of the wait
  - `wait_until`: another timed version
  - `notify`: unblocks the first thread in the queue
  - `notify_all`: unblocks all threads in the queue
  - `wait(ulock, pred)` is a shortcut for `while (pred) wait(ulock);`
- The while is necessary because of spurious wake-ups
  - It may happen that the thread is unblocked, but the condition is false

The mailbox example

- The following is a didactic example to demonstrate all we have see until now
- See `examples/mailbox.hpp` and `examples/thread_mailbox.cpp`

Synchronization barrier

- Let us write a synchronization barrier
  - threads can continue execution only when all thread reach the barrier
- See `examples/thread_group.cpp`

Outline

Asynchronous calls

- In C++ it is possible to defer the call to a function to another thread, or later on when the result is actually needed (lazy evaluation).
  - this is done through the async function

```
template< class Function, class... Args>
std::future<typename std::result_of<Function(Args...)>::type>
    async( Function&& f, Args&&... args );

template< class Function, class... Args >
std::future<typename std::result_of<Function(Args...)>::type>
    async( std::launch policy, Function&& f, Args&&... args );
```

- Launch policy can be
  - `std::launch::async`: a thread is created
  - `std::launch::deferred`: lazy evaluation
  - If both are specified in or, the actual behaviour is implementation dependent

Asynchronous calls and futures

- An async call returns a `future`, that is an object that at some point will contain the result of the function
- If a thread tries to obtain the value before the function has completed, it blocks
  - see `examples/parallel_sum.cpp`

Async and futures

- Warning: if the future returned from async is not copied into a variable, the destructor waits for the function to terminate
- In other words, it does not parallelise anything

```
std::async(std::launch::async, []{ return f(); });
// temporary's dtor waits for f()
std::async(std::launch::async, []{ return g(); });
// does not start until f() completes
```

Future validity

- When you call `get` on a future, you are actually moving out the result from the future
- After calling `get`, the future becomes *invalid*
- To check the validity of a future, you must call `valid`
- If you call `get` on an invalid future, the behaviour is undefined
  - most implementation will throw a `std::future_error` exception

Packaged task

- A packaged task is similar to an async,
- it is an object that wraps a function, that can be later executed in several ways:
  - as a thread
  - as a normal function
- we can obtain the corresponding future with `get_future`

Example of packaged task into a thread

- See http://en.cppreference.com/w/cpp/thread/packaged_task

Outline

Parallel code

- It is quite difficult to write fast parallel code, because of the many sharing parts
- Let's try to write a very simple parallel program, and let's see if we can make it fast
- We want to compute how many elements in a matrix are odd.
- Sequential code:

```
int odds = 0;
for( int i = 0; i < DIM; ++i )
  for( int j = 0; j < DIM; ++j )
    if( matrix[i*DIM + j] % 2 != 0 )
       ++odds;
```

Naive parallel code

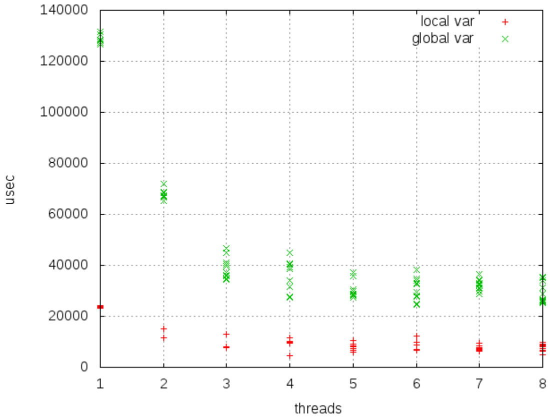
```
void body(unsigned start, unsigned end, int &result) {
    for (unsigned i=start; i<end; ++i) {
        for (unsigned j=0; j<NELEM; ++j)
          if (matrix[i][j] % 2) ++result;
    }
}
...
vector<int> result(nthreads);
for (unsigned p = 0; p<nthreads; ++p) {
    ...
    th.emplace_back(thread(body, start, end, result[p]));
}
total = accumulate(result.begin(), result.end(), 0);
```

Analysis

- If all threads write on global variables there can be a lot of load on the cache.
- It may be better to operate on local variables, and then write the result only at the end

```
void body(unsigned start, unsigned end, int &result)
{
    int temp = 0;
    for (unsigned i=start; i<end; ++i)
        for (unsigned j=0; j<NELEM; ++j)
            if (matrix[i][j] % 2) ++temp;
    result = temp;
}
```

Comparison



local var
global var
140000
120000
100000
80000
60000
40000
20000
0
usec
1
2
3
4
5
6
7
8
threads

Conclusions

- Effective parallelization is not so easy
  - the structure of the actual hw architecture is very important
  - L1 and L2 cache size
  - number of cores, number of hyperthreads
  - brach prediction
  - cache pre-load algorithms
  - size of the data structures (do they fit in the cache?)
  - compiler
  - etc.
- Only approach: try different solutions and and measure!