Threads

P. Ronchese
Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

Thread definition

Thread: set of operations executed sequentially

Traditionally, all operations in a program are executed in one single thread, but:

- The optimization of the processor capabilities can require the execution of more threads at one time.
- Two possible ways of executing more threads:
  - the threads are executed on the same processor,
  - the threads actually share the time.
- Multi-core processors and multi-processor computers allow the actual execution of more operations at the same time.

Threads in C++

C++11 only

The simplest instruction to execute a function in a separate thread is the instantiation of a `std::thread` object.
To use it an `#include <thread>` is required.

```
#include <thread>
void f(...);
  ...
  std::thread t(f,...); // start a thread
  ...
  t.join(); // wait for thread completion
  ...       // if not yet finished
```

In the compilation with `gcc`
the optional argument `-pthread` must be given.

Thread arguments

Several objects can be used to start a thread:

- a global function,
- a “lambda” function, i.e. a function coded inside the instruction referring to it (in this case the thread creation),
- a functor, i.e. the instance of a class with the `()` operator.

Function arguments

The arguments required by the function (or functor) are to be listed after the function (or functor) name in the thread instantiation.

Access to data

More than one thread can access to the same variable or object, at the same time or at different times.

- The compiler can be not aware that a variable is modified by another thread, and produce wrong code in optimization attempt: `volatile` keyword prevents this.
- A resurce can be accessible by one thread only at a time, multiple access can be prevented by mean of locks.

```
volatile int i; // "i" is accessed by
                // other threads
int j=100;
while(j--) {
  i=10; // "i" must be set at 10 at each
  ...   // iteration, even if not changed
}
```

Data race

When more than one thread accesses one variable, and at least one of them tries to modify it, a “data race” occurs.

```
...
i=12;
j=25;
int k=i+j; // "k" may differ from 37
```

Locks

A “mutually exclusive lock” is an object the control of which can be taken by at most one thread at a time.

```
std::mutex m;
  ...
  m.lock();
  ...// this code can be executed by
  ...// only one thread at a time
  m.unlock();
```

Race condition

A “race condition” is the situation where the result of an operation depends on other concurrent operations.

Threads returning a value

- A thread producing a result can put it in a `std::promise` object.
- The result can be retrieved through an associated `std::future` object.
- The progress status can be inspected.
- When the `std::future` object is queried the execution wait for the completion of the thread setting it.

Status inspection and result retrieve

```
#include <future>
void func(...,std::promise<int> i) {
   ...
   i.set(...);
}
   ...
   std::promise<int> pi;
   std::future <int> fi=pi.get_future();
   std::thread t(func,...,std::move(pi));
   ...
   while(fi.wait_for(std::chrono::seconds(1)))
         cout<<"waiting..."<<endl;
   int i=fi.get();
```

Direct `std::future` retrieval

An explicit `std::promise` set can be avoided using a `std::pakaged_task` object.

```
#include <future>
int func(...)  {
   ...
   return ...;
}
   ...
   std::packaged_task<int(...)> pt(func);
   std::future<int> fi=pt.get_future();
   std::thread t(std::move(pt),...);
   ...
   int i=fi.get();
```

Single instruction start

The thread can be started and the associated `future` retrieved with a single call to `std::async`.

```
#include <future>
int func(...)   {
   ...
   return ...;
}
   ...
   std::future<int> fi=
     std::async(std::launch::async,func,...);
   ...
   int i=fi.get();
```