POLITECNICO DI MILANO

Advanced Operating Systems

Multi-threading in C++

Giuseppe Massari, Federico Terraneo

giuseppe.massari@polimi.it, federico.terraneo@polimi.it

Outline

Introduction

- Multi-tasking vs multi-threading

C++11 threading support

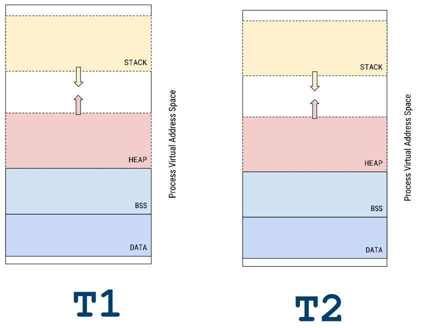
- Thread creation
- Synchronization
- Mutex usage and pitfalls
- Condition variable

Design patterns

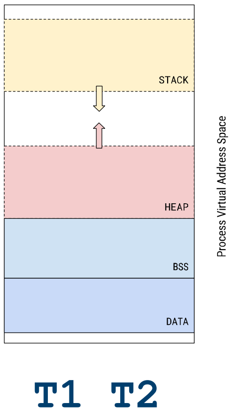
- Producer/Consumer
- Active Object
- Reactor
- ThreadPool

Multi-tasking vs multi-threading

- Multi-tasking operating systems allow to run more “processes” concurrently
  - A single application can spawn multiple processes
  - OS assigns a separate address space to each process (a process cannot directly access the address space of another process)



- Multi-threading allows a single process to perform multiple tasks concurrently (in a shared address space)
  - Fast and easy sharing of data structures among tasks



- Both were introduced to improve performance and responsiveness of computing systems
- *This class is focused on multi-threading*

Why multi-threading?

- Allowing an applications to perform multiple tasks concurrently

  For example running a responsive user interface, handling network communications and running heavy batch computations at the same time

- Parallelizing algorithms in order to exploit multi-core CPU

  This became popular with the widespread adoption of multi-core architectures

Multi-threading support

- HW side: we are currently in the multi-core (and many-core) era

  Additional performance of computing architectures delivered through an increasing number of cores

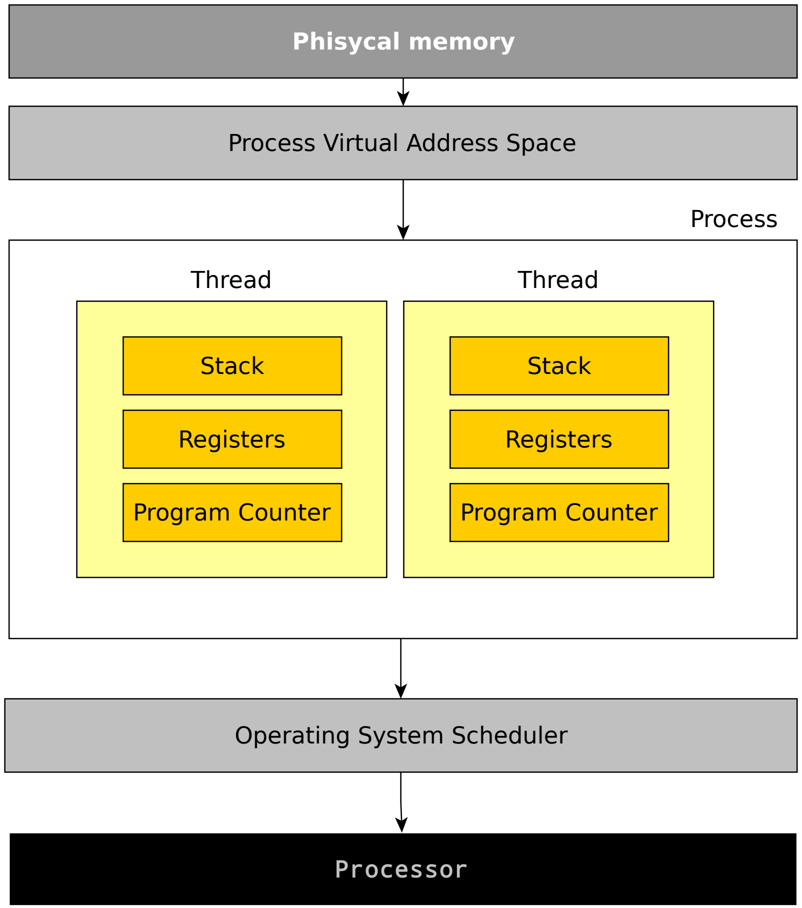
- SW side: Growing importance in the computing landscape

  Programming languages are adding native support for multi-threading

  Example: C++ starting from C++11 standard version

Thread

- A thread is defined as a "lightweight task"
- Each thread has a separate stack and context (registers and program counter value)
- Depending on the implementation the OS or the language runtime are responsible of the thread-to-core scheduling



C++11 multi-threading support

Class std::thread

- Constructor starts a thread executing the given function

```
#include <iostream>
#include <thread>
using namespace std;
using namespace std::chrono;

void myThread() {
    for(;;) {
        cout<<"world "<<endl;
        this_thread::sleep_for(milliseconds(500));
    }
}
int main() {
    thread t(myThread);
    for(;;) {
        cout<<"hello"<<endl;
        this_thread::sleep_for(milliseconds(500));
    }
}
```

C++11 multi-threading support

Class std::thread

```
$ g++ main.cpp -o test -std=c++11 -pthread
```

- There is no guarantee about the threads execution order

```
$ ./test
helloworld

helloworld

hello
world
helloworld

hello
world
helloworld
```

Class `std::thread`

- Thread constructor can take additional arguments that are passed to the thread function
- **`join()`** member function waits for the thread to complete

```
#include <iostream>
#include <thread>
using namespace std;


void myThread(int i, const string & s) {
   cout<<"Called with "<<i<<" and "<<s<<endl;
}

int main() {
   thread t(myThread, 2, "test");
   t.join();
}
```

C++11 multi-threading support

Synchronization

- What is the output of the following code?

```
#include <iostream>
#include <thread>
using namespace std;
static int sharedVariable=0;

void myThread() {
   for(int i=0;i<1000000;i++) sharedVariable++;
}

int main() {
   thread t(myThread);
   for(int i=0;i<1000000;i++) sharedVariable--;
   t.join();
   cout<<"sharedVariable="<<sharedVariable<<endl;
}
```

Synchronization

```
$ ./test
sharedVariable=-313096
$ ./test
sharedVariable=-995577
$ ./test
sharedVariable=117047
$ ./test
sharedVariable=116940
$ ./test
sharedVariable=-647018
```

- Accessing the same variables from different threads can lead to *race conditions*
- The two threads are concurrently changing the value of the same variable

Synchronization

- What does happen under the hood?

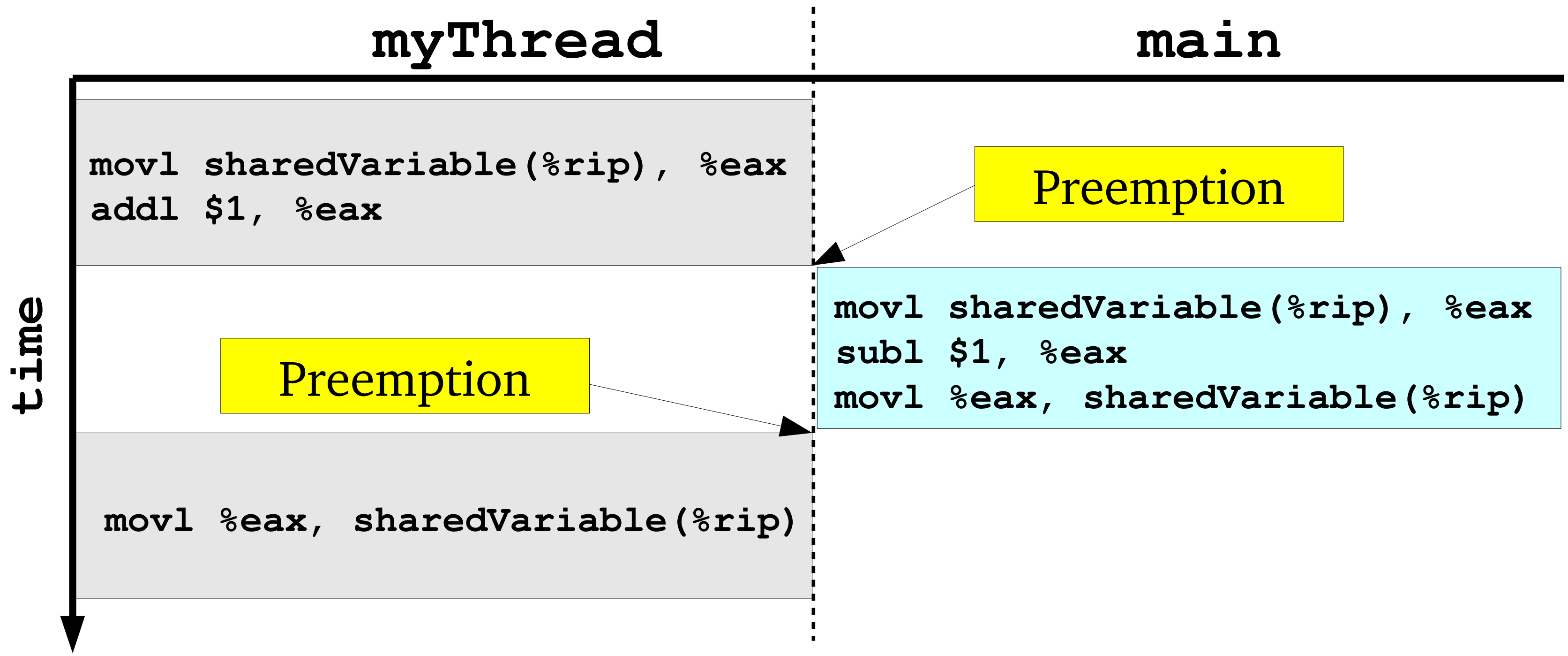
```
 //sharedVariable++
movl   sharedVariable(%rip), %eax
addl   $1, %eax
movl   %eax, sharedVariable(%rip)

//sharedVariable--
movl   sharedVariable(%rip), %eax
subl   $1, %eax
movl   %eax, sharedVariable(%rip)
```

- Increment (++) and decrement (–) are not *atomic* operations
- The operating system can preempt a thread between any instruction

Synchronization

- What does happen under the hood?



- **`MyThread`** has been preempted before the result of the increment operation has been written back (**`sharedVariable`** update)
- This leads to incorrect and unpredictable behaviours

Synchronization

- A *critical section* is a sequence of operations accessing a shared data structure that must be performed atomically to preserve the program correctness
- In the example we faced a race condition, since the two threads entered a critical section in parallel
- To prevent race conditions we need to limit concurrent execution whenever we enter a critical section

C++11 multi-threading support

Solution 1: *Disable preemption*

- Dangerous – it may lock the operating system
- Too restrictive – it is safe to preempt to a thread that does not modify the same data structure
- Does not work on multi-core processors

Solution 2: *Mutual exclusion*

- Before entering a critical section a thread checks if it is “free”
  - If it is, enter the critical section
  - If not, it blocks
- When exiting a critical section a thread checks if there are other blocked threads
  - If yes, one of them is selected and woken up

Class `std::mutex`

- It has two member functions
  - `lock()` to call before entering a critical section
  - `unlock()` to call after exiting a critical section

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;
static int sharedVariable=0;
mutex myMutex;

void myThread() {
    for(int i=0;i<1000000;i++) {
        myMutex.lock();
        sharedVariable++;
        myMutex.unlock();
    }
}
```

```
int main() {
    thread t(myThread);
    for(int i=0;i<1000000;i++) {
        myMutex.lock();
        sharedVariable--;
        myMutex.unlock();
    }
    t.join();
    cout<<"sharedVariable="
        <<sharedVariable<<endl;
}
```

Deadlock

- Improper use of mutex may lead to *deadlock,* according to which program execution get stuck
    - Deadlocks may occur due to several causes
- Cause 1: *Forgetting to unlock a mutex*

```
…
mutex myMutex;
int sharedVariable;

void myFunction(int value) {
    myMutex.lock();
    if(value<0) {
        cout<<"Error"<<endl;
        return;
    }
    SharedVariable += value;
    myMutex.unlock();
}
```

A function returns without unlocking the previously locked mutex

Next function call will result in a deadlock

Deadlock

- Solution: C++11 provides *scoped lock* that automatically unlocks mutex, regardless of how the scope is exited

```
…
mutex myMutex;
int sharedVariable;

void myFunction(int value) {
    {
        lock_guard<mutex> lck(myMutex);
        if(value<0)
        {
            cout<<"Error"<<endl;
            return;
        }
        SharedVariable += value;
    }
}
```

myMutex unlocked here

OR

myMutex unlocked here

Deadlock

- Cause 2: *Nested function calls locking the same mutex*

```
…
mutex myMutex;
int sharedVariable;

void func2()
{
    lock_guard<mutex> lck(myMutex);
    doSomething2();
}

void func1()
{
    lock_guard<mutex> lck(myMutex);
    doSomething1();
    func2();
}
```

Deadlock if we're called by **`func1()`**

Called with the mutex locked

C++11 multi-threading support

Deadlock

- Solution: *Recursive mutex* permit multiple locks by the *same thread*

```
…
recursive_mutex myMutex;
int sharedVariable;

void func2() {
    lock_guard<recursive_mutex> lck(myMutex);
    doSomething2();
}

void func1(){
    lock_guard<recursive_mutex> lck(myMutex);
    doSomething1();
    func2();
}
```

- However recursive mutex are more expensive than mutex

  Use them only when needed

Deadlock

- Cause 3: *Order of locking of multiple mutexes*

```
…
mutex myMutex1;
mutex myMutex2;

void func2() {
    lock_guard<mutex> lck1(myMutex1);
    lock_guard<mutex> lck2(myMutex2);
    doSomething2();
}

void func1(){
    lock_guard<mutex> lck1(myMutex2);
    lock_guard<mutex> lck2(myMutex1);
    doSomething1();
}
```

- Thread1 calls **`func1()`**, locks **`myMutex2`** and blocks on **`myMutex1`**
- Thread2 calls **`func2()`**, locks **`myMutex1`** and blocks on **`myMutex2`**

Deadlock

- Solution: C++11 **lock** function takes care of the correct locking order

```
mutex myMutex1;
mutex myMutex2;

void func2() {
    lock(myMutex1,myMutex2);
    doSomething2();
    myMutex1.unlock();
    myMutex2.unlock();
}
void func1(){
    lock(myMutex2,myMutex1);
    doSomething1();
    myMutex2.unlock();
    myMutex1.unlock();
}
```

- Any number of mutexes can be passed to **lock** and in any order

    Use of **lock** is more expensive of **lock_guard**

C++11 multi-threading support

Deadlocks and race conditions

- Are faults that occur due to some “unexpected” order of execution of the threads
  - Correct programs should work regardless of the execution order
- The order that triggers the fault can be **extremely** uncommon
  - Running the same program million of times may still not trigger the fault
- Are thus hard to debug
  - It is difficult to reproduce the bug
- Testing is almost useless for checking such errors
  - Good design is mandatory

Loosing concurrency

- Leaving a mutex locked for a long time reduces the concurrency in the program

```
…

mutex myMutex;
int sharedVariable=0;

void myFunction()
{
    lock_guard<mutex> lck(myMutex);
    sharedVariable++;
    this_thread::sleep_for(milliseconds(500));
}
```

Loosing concurrency

- Solution: Keep critical sections as short as possible
  - Leave unnecessary operations out of the critical section

```
…
mutex myMutex;
int sharedVariable=0;

void myFunction()
{
    {
        lock_guard<mutex> lck(myMutex);
        sharedVariable++;
    }
    this_thread::sleep_for(milliseconds(500));
}
```

Condition variable

- In many multi-threaded programs we may have dependencies among threads
- A “dependency” can come from the fact that the thread must wait for another thread to complete its current operation
- In such a case we need a mechanism to explicitly block a thread

class `std::condition_variable`

- Three member function

  `wait(unique_lock<mutex> &)` - block the thread until another thread wakes it up. The mutex is unlocked for the duration of the **`wait(...)`**

  `notify_one()` - wake up one of the waiting threads

  `notify_all()` - wake up all the waiting threads

  - If no thread is waiting do nothing

class `std::condition_variable`

- In the example, **`myThread`** is waiting for the **`main`** to complete the read from standard input

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
using namespace std;
string shared;
mutex myMutex;
condition_variable myCv;

void myThread() {
    unique_lock<mutex> lck(myMutex);
    if(shared.empty()) myCv.wait(lck);
    cout<<shared<<endl;
}
```

```
int main() {
    thread t(myThread);
    string s;
    cin>>s; // read from stdin
    {
        unique_lock<mutex>
            lck(myMutex);
        shared=s;
        myCv.notify_one();
    }
    t.join();
}
```

Outline

Introduction

- Multi-tasking vs multi-threading

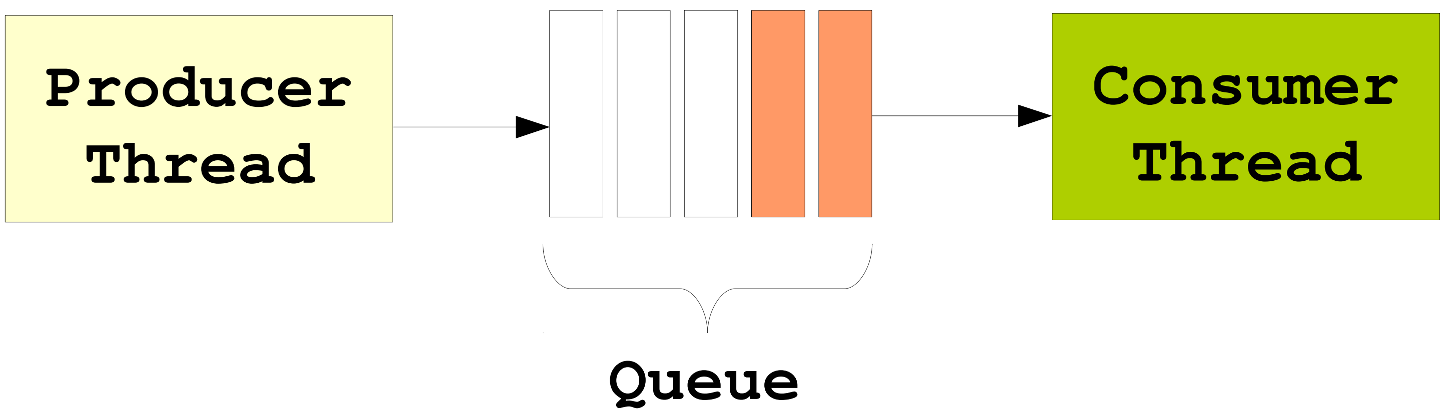
C++11 threading support

- Thread creation
- Synchronization
- Mutex usage and pitfalls
- Condition variable

Design patterns

- Producer/Consumer
- Active Object
- Reactor
- ThreadPool

Producer/Consumer



- A thread (consumer) needs data from another thread (producer)
- To decouple the operations of the two threads we put a queue between them, to buffer data if the producer is faster than consumer
- The access to the queue needs to be synchronized
  - Not only using a mutex but the consumer needs to wait if the queue is empty
  - Optionally, the producer may block if the queue is full

Producer/Consumer

- synchronized_queue.h (1/2)

```
#ifndef SYNC_QUEUE_H_
#define SYNC_QUEUE_H_
#include <list>
#include <mutex>
#include <condition_variable>
template<typename T>
class SynchronizedQueue {
public:
    SynchronizedQueue(){}
    void put(const T & data);
    T get();
private:
    SynchronizedQueue( const SynchronizedQueue &);
    SynchronizedQueue & operator=(const SynchronizedQueue &);
    std::list<T> queue;
    std::mutex myMutex;
    std::conditionVariable myCv;
};
…
```

Producer/Consumer

- **synchronized_queue.h (2/2)**

```
template<typename T>
void SynchronizedQueue<T>::put(const T& data)
{
    std::unique_lock<std::mutex> lck(myMutex);
    queue.push_back(data);
    myCv.notify_one();
}

template<typename T>
T SynchronizedQueue<T>::get()
{
    std::unique_lock<std::mutex> lck(myMutex);
    while(queue.empty())
        myCv.wait(lck);
    T result=queue.front();
    queue.pop_front();
    return result;
}
#endif // SYNC_QUEUE_H_
```

Producer/Consumer

- **main.cpp**

```
#include "synchronized_queue.h"
#include <iostream>
#include <thread>
using namespace std;
using namespace std::chrono;

SynchronizedQueue<int> queue;

void myThread() {
    for(;;) cout<<queue.get()<<endl;
}
int main() {
    thread t(myThread);
    for(int i=0;;i++) {
        queue.put(i);
        this_thread::sleep_for(seconds(1));
    }
}
```

Producer/Consumer

- What if we do not use the condition variable?
- **synchronized_queue.h (1/2)**

```
#ifndef SYNC_QUEUE_H_
#define SYNC_QUEUE_H_
#include <list>
#include <mutex>
template<typename T>
class SynchronizedQueue {
public:
    SynchronizedQueue(){}
    void put(const T & data);
    T get();
private:
    SynchronizedQueue( const SynchronizedQueue &);
    SynchronizedQueue & operator=(const SynchronizedQueue &);
    std::list<T> queue;
    std::mutex myMutex;
//  std::conditionVariable myCv;
};
…
```

Producer/Consumer

- **synchronized_queue.h (2/2)**

```
template<typename T>
void SynchronizedQueue<T>::put(const T & data)
{
    std::unique_lock<std::mutex> lck(myMutex);
    queue.push_back(data);
    //myCv.notify_one();
}

template<typename T>
T SynchronizedQueue<T>::get() {
    for(;;) {
        std::unique_lock<std::mutex> lck(myMutex);
        if(queue.empty()) continue;
        T result=queue.front();
        queue.pop_front();
        return result;
    }
}
#endif // SYNC_QUEUE_H_
```

Producer/Consumer

- What if we do not use the condition variable?
- The consumer is left “spinning” when the queue is empty
  - This takes up precious CPU cycles and slows down other threads in the system
  - Keeping the CPU busy increases power consumption
- Although the code is correct from a functional point of view this is a bad programming approach
  - When a thread has nothing to do it should block to free the CPU for other threads and reduce power consumption
- Extension: *Try to implement the version with a limited queue size*
  - The producer shall block if the queue reaches the maximum size

Active Object

- To instantiate “task objects”
- A thread function has no explicit way for other threads to communicate with it
  - Often data is passed to thread by global variables
- Conversely, this pattern allows us to wrap a thread into an object, thus having a “thread with methods you can call”
  - We may have member functions to pass data while the task is running, and collect results
- In some programming languages (e.g., Smaltalk and Objective C) all objects are “active objects”

Active Object

- The class includes a thread object and a member function **run()** implementing the task

```
#ifndef ACTIVE_OBJ_H_
#define ACTIVE_OBJ_H_
#include <atomic>
#include <thread>
class ActiveObject {
public:
    ActiveObject();
    ~ActiveObject();
private:
    virtual void run();
    ActiveObject(const ActiveObject &);
    ActiveObject& operator=(const ActiveObject &);
protected:
    std::thread t;
    std::atomic<bool> quit;
};
#endif // ACTIVE_OBJ_H_
```

Active Object

```
#include "active_object.h"
#include <chrono>
#include <functional>
#include <iostream>
using namespace std;
using namespace std::chrono;

ActiveObject::ActiveObject():
    t(bind(&ActiveObject::run, this)), quit(false) {}

void ActiveObject::run() {
    while(!quit.load()) {
        cout<<"Hello world"<<endl;
        this_thread::sleep_for(milliseconds(500));
    }
}
ActiveObject::~ActiveObject() {
     if(quit.load()) return; //For derived classes
    quit.store(true);
    t.join();
}
```

Active Object

- The constructor initialize the thread object, while the destructor takes care of joining it
- The **`run()`** member function acts as a “main” concurrently executing
- Commonly used to implement threads communicating through a producer/consumer approach
- In the example implementation we used the “atomic” variable **`quit`** to terminate the **`run()`** function when the object is destroyed
  - A normal boolean variable with a mutex would work as well

bind and function

- C has no way to decouple function *arguments binding* from the call
- In C++11 **bind** and **function** allow us to package a function and its arguments, and call it later

```
#include <iostream>
#include <functional>
using namespace std;

void printAdd(int a, int b){
    cout<<a<<'+'<<b<<'='<<a+b<<endl;
}

int main() {
    function<void ()> func;
    func = bind(&printAdd, 2, 3);
    …
    func();
}
```

We want to handle a function as if it was an object

We specify the function arguments without performing the call

Function call (with already packaged arguments)

Reactor

- The goal is to decouple the task creation from the execution
- A executor thread waits on a task queue
- Any other part of the program can push tasks into the queue
- Task are executed sequentially
  - The simplest solution is usually in a FIFO order
  - We are free to add to the “reactor” alternative thread scheduling functions
- C++11 **`bind`** and **`function`** allows us to create the task, leaving the starting time to the executor thread, in a second step

Reactor

- The class derives from ActiveObject to implement the executor thread and uses the SynchronizedQueue for the task queue

```
#ifndef REACTOR_H_
#define REACTOR_H_
#include <functional>
#include "synchronized_queue.h"
#include "active_object.h"

class Reactor: public ActiveObject {
public:
    void pushTask(std::function<void ()> func);
    virtual void ~Reactor();
private:
    virtual void run();
    SynchronizedQueue<std::function<void ()>> tasks;
};
#endif // REACTOR_H_
```

Reactor

```
#include "reactor.h"
using namespace std;

void doNothing() {}

void Reactor::pushTask(function<void ()> func) {
    tasks.put(func);
}

Reactor::~Reactor() {
    quit.store(true);
    pushTask(&doNothing);
    t.join(); // Thread derived from ActiveObject
}

void Reactor::run() {
    while(!quit.load())
        tasks.get()(); // Get a function and call it
}
```

Reactor

- In the example we are pushing a task to execute the **`printAdd()`** function

```
#include <iostream>

using namespace std;

void printAdd(int a, int b)
{
    cout<<a<<'+'<<b<<'='<<a+b<<endl;
}

int main()
{
    Reactor reac;
    reac.pushTask(bind(&printAdd,2,3));
    …
}
```

ThreadPool

- The limit of the Reactor pattern is due to the fact that tasks are processed sequentially
  - The latency of the task execution is dependent on the length of the task queue
- To reduce latency and exploit multi-core computing architectures we can have multiple executor threads waiting on the same task queue
- *Try to implement your own ThreadPool at home!*