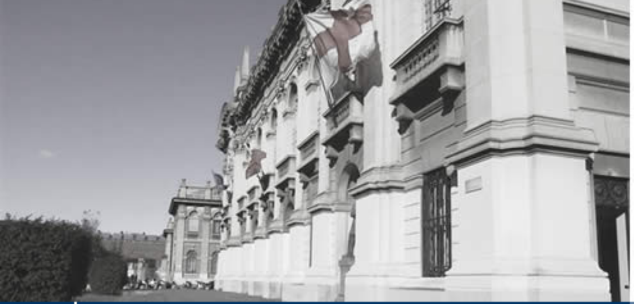


POLITECNICO DI MILANO





Multi-threaded Programming in C++

Advanced Operating Systems

Giuseppe Massari, Federico Terraneo
giuseppe.massari@polimi.it
federico.terraneo@polimi.it

Outline

Introduction

- Multi-tasking implementations

C++11 threading support

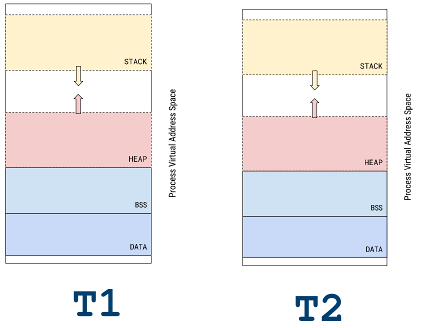
- Thread creation
- Synchronization
- Mutual exclusion and pitfalls
- Condition variables
- Task-based approaches

Design patterns

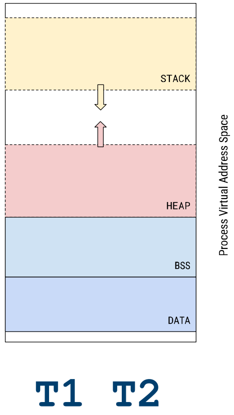
- Producer/Consumer
- Active Object
- Reactor
- ThreadPool

Multi-tasking implementations

- Multi-tasking operating systems allow to run more “tasks” concurrently
- *Multi-process* implementation
  - 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* implementation
  - A single application spawn multiple **threads**
  - Fast and easy sharing of data structures among tasks
    - The address space is shared among threads



Introduction

Why multi-tasking?

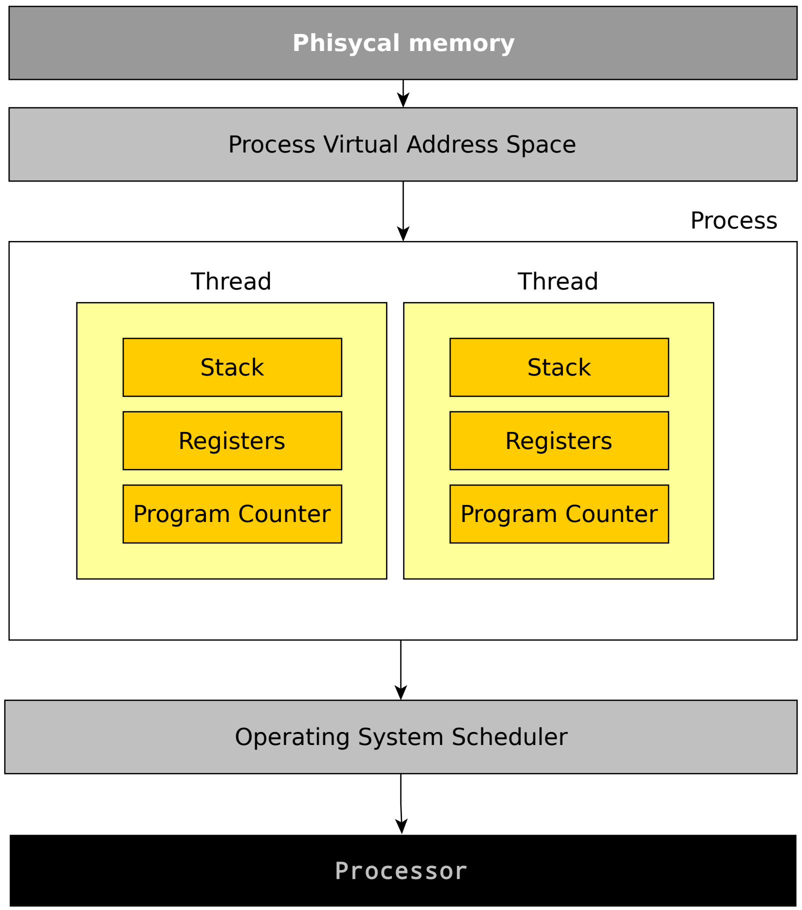
- Improving *performance* and *responsiveness* of a system/application
- *Task parallelism*
  - Single application performing multiple different tasks concurrently
- *Data parallelism*
  - Single application running the same task on different chunk of data

Multi-threading support

- HW side: we are currently in the multi-core (and many-core) era
  - Increasing number of cores to exploit
- 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 (Lightweigth Process – LWP- in Linux)*
- 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

Thread

- C++11 introduced the class **thread** (namespace **std**)
  - Definition: **#include <thread>**

Member function	Return value	Description
`get_id()`	`thread::id`	Returns an unique identifier object
`detach()`	`void`	Allows the thread to run independently from the others
`join()`	`void`	Blocks waiting for the thread to complete
`joinable()`	`bool`	Check if the thread is joinable
`hardware_concurrency()`	`unsigned`	An hint on the HW thread contexts (often, number of CPU cores)
`operator=`		Move assignment

Thread

- C++11defined namespace **std::this_thread** to group a set of functions to access the current thread

Member function	Return value	Description
`get_id()`	`thread::id`	Returns an unique identifier object for the current thread
`yield()`	`void`	Suspend the current thread, allowing other threads to be scheduled to run
`sleep_for()`	`void`	Sleep for a certain amount of time
`sleep_until()`	`void`	Sleep until a given timepoint

Thread

- Example: *Hello world*

```
#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));
    }
}
```

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
```

Thread

- Thread constructor can take additional arguments that are passed to the thread function

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

void myFunc(const string & s) {
    for(;;) {
        cout << s <<endl;
        this_thread::sleep_for(milliseconds(500));
    }
}
int main() {
    thread t(myFunc, “world”);
    myFunc(“hello”);
}
```

Thread

- Example: *Inter-thread synchronization*

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

void myFunc(const string & s) {
    for(int i=0; i<10; ++i) {
        cout<< s <<endl;
        this_thread::sleep_for(milliseconds(500));
    }
}
int main() {
    thread t(myFunc, "world");
    myFunc("hello");
    if (t.joinable())
        t.join();
}
```

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
```

- We expect sharedVariable to be equal 0, since the two threads increment and decrement respectively iterating for the same amount of cycles
- To understand where the issue comes from we must observe the --/++ statements at the assembly level

Synchronization

- What does happen under the hood?

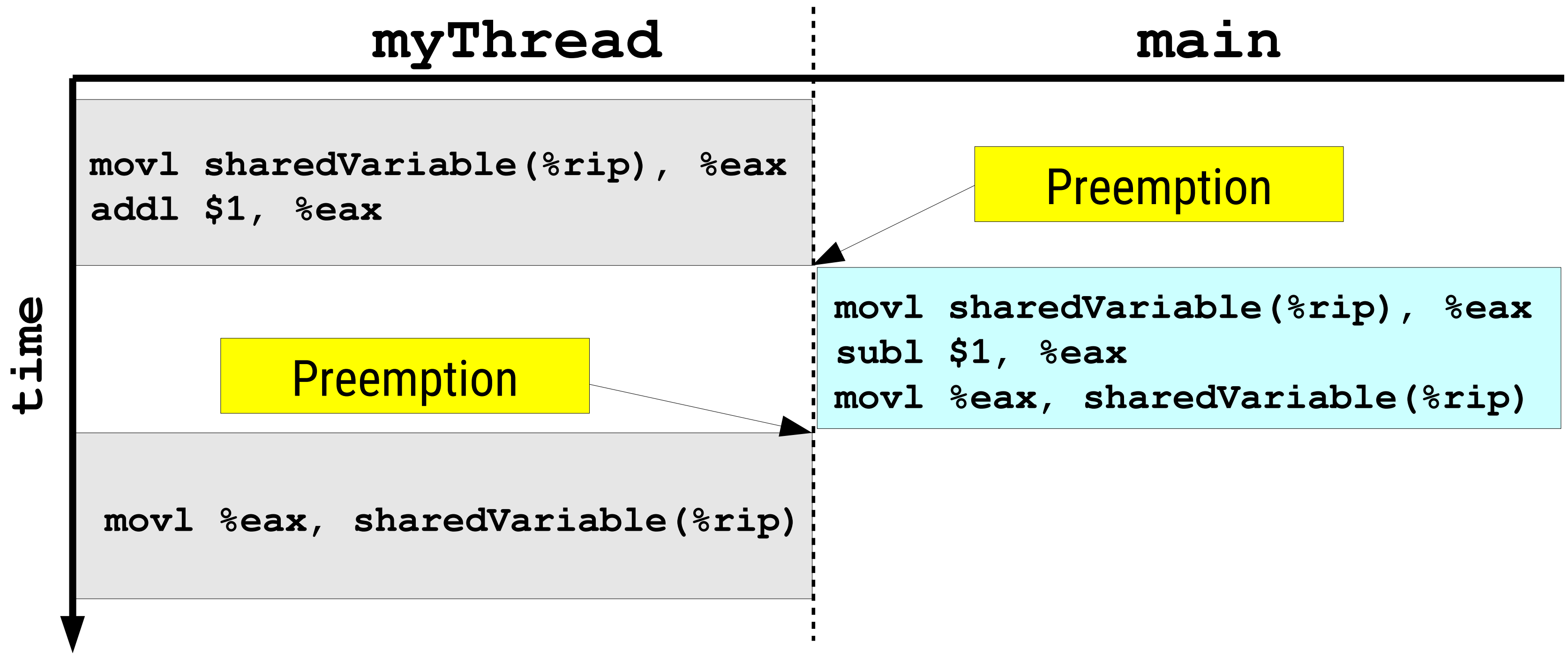
```
 //sharedVariable++
movl   sharedVariable(%rip), %eax
addl   $1, %eax
movl   %eax, sharedVariable(%rip)

//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 instructions

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 is a *race condition*, leading to incorrect and unpredictable behaviours

C++11 multi-threading support

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

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
    - Otherwise 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

Mutex

- The mutex is a widespread synchronization primitive provided to perform mutual exclusive access to data
- C++11 introduced the class **mutex** (namespace **std**)
  - Definition: **#include <mutex>**

Member function	Description
`lock()`	Locks the mutex. If already locked, the thread blocks.
`try_lock()`	Try to lock the mutex. If already locked, returns.
`unlock()`	Unlock the mutex.

Mutex

- Example: simple protection of a critical section

```cpp
#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();
    }
}
```

```cpp
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

- Improper use of mutex may lead to *deadlock,* according to which program execution get stuck
  - Deadlocks may occur due to several causes
- Cause 2: *Unexpected function termination*

```
…
mutex myMutex;
int sharedVariable;

void myFunction(int value) {
    myMutex.lock();

    int var = new int;
    ...
    SharedVariable += value;
    myMutex.unlock();
}
```

Code throwing exceptions (as ‘new’ could do) in another condition for which function may exit, leaving the mutex locked

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 3: *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* → 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 a recursive mutex introduces higher overhead compared to a standard mutex

Deadlock

- Cause 4: *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 runs **`func1()`**, locks **`myMutex2`** and blocks on **`myMutex1`**
- Thread2 runs **`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);
    lock_guard<mutex> lk1(myMutex1, adopt_lock);
    lock_guard<mutex> lk2(myMutex2, adopt_lock);
    doSomething2();
}
void func1(){
    lock(myMutex2, myMutex1);
    lock_guard<mutex> lk1(myMutex1, adopt_lock);
    lock_guard<mutex> lk2(myMutex2, adopt_lock);
    doSomething1();
}
```

- Any number of mutexes can be passed to **`lock`** and in any order
  - Use of **`lock`** is more expensive than **`lock_guard`**

Deadlocks and race conditions

- Faults occurring due to “unexpected” order of execution of 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
- Multi-threaded programs are 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++;
    cout << “The current value is: ” << 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()
{
    int temp;
    {
        lock_guard<mutex> lck(myMutex);
        temp = ++sharedVariable;
    }
    cout << “The current value is: ” << temp;
    this_thread::sleep_for(milliseconds(500));
}
```

Condition variables

- In many multi-threaded programs we may have dependencies among threads

- A “dependency” can come from the fact that a thread must wait for another one to complete its current operation
  - This waiting must be unexpensive, i.e., the thread must possibly consume no CPU time, since not performing any useful work

- In such a case we need a mechanism to...
  - Explicitly block a thread
  - Put the thread into a waiting queue
  - Notify the thread when the condition leading to its block has changed

Condition variables

- C++11 introduced class **condition_variable** (namespace **std**)
  - Definition: **#include <condition_variable>**

Member function	Description
`wait(unique_lock<mutex> &)`	Blocks the thread until another thread wakes it up. The Lockable object is unlocked for the duration of the `wait(...)`
`wait_for(unique_lock<mutex> &, const chrono::duration<..> t)`	Blocks the thread until another thread wakes it up, or a time span has passed.
`notify_one()`	Wake up one of the waiting threads.
`notify_all()`	Wake up all the waiting threads. If no thread is waiting do nothing.

Condition variables

- Example: ***`myThread`*** *is waiting for* ***`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 myFunc() {
    unique_lock<mutex> lck(myMutex);
    while(shared.empty())
        myCv.wait(lck);
    cout<<shared<<endl;
}
```

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

Task-based parallel programming

- Sometimes we need to run *asynchronous tasks* producing output data that will become useful later on…
  - Thread objects are OK for that but.. what about getting a return value from the executed function?

- We saw the basics of *thread-based* parallel programming but in C++11 we can talk also about *task-based* parallel programming
  - It relies on a different constructs (no **std::thread** objects)
  - It enables the possibility of handling return values

Future

- C++11 introduced class **future** to access values set values from specific providers
  - Definition: **#include <future>**
  - Providers: calls to **async()**, objects **promise<>** and **packaged_task<>**
  - Providers set the *shared state* to ready when the value is set

Member function	Return value	Description
`operator=`		Move assignment
`get()`	T	Returns the stored value if the future is ready. Blocks, i not ready.
`valid()`	`bool`	Once the shared state is retrieved with get(), this function returns false.
`wait()`	`void`	Blocks until the shared state is set to ready.
`wait_for()`	`void`	Blocks until the shared state is set to ready or a time span has passed.

Future providers

- C++11 introduced function **async** (namespace **std**)
  - Definition: **#include <future>**
  - Higher level alternative to **`std::thread`** to execute functions in parallel

```
future<T> async(launch_policy, function, args...);
```

- T is the return type of the function
- Three different launch policies for spawning the task

Policy	Description
`launch::async`	*Asynchronous*: Launches a new thread to call function
`launch::deferred`	The call to function is deferred until the shared state of the *future* is accessed (call to **wait** or **get**)
`launch::async | launch::deferred`	System and library implementation dependent. Choose the policy according to the current availability of concurrency in the system.

C++11 multi-threading support

Future providers

- Example: *Basic async() usage*

```
#include <future>
#include <iostream>

// function to check if a number is prime
bool is_prime (int x) { … }

int main () {
  std::future<bool> fut = std::async(
                  std::launch::async, is_prime, 117);
  // … do other work …

  bool ret = fut.get();  // waits for is_prime to return
  return 0;
}
```

- `fut.get()` blocks until `is_prime()` returns

C++11 multi-threading support

Future providers

- C++11 introduced a further facility, the class **std::packaged_task<>**
- This class wraps a *callable element* (e.g. a function pointer) and allows to retrieve asynchronously its return value

```
std::packaged_task<function_type> tsk(args);
```

Member function	Return value	Description
`operator=`		Move assignment
`operator()`	T	Call stored function
`valid()`	`bool`	Check for the shared state to be valid
`get_future()`	`future<T>`	Get the function object
...		

Future providers

- Example: *Basic packaged_task<> usage*

```
#include <future>
...
using namespace std;
int compute_double(int value) { return value*2; }

int main() {
    packaged_task<int(int)> tsk(compute_double);
    future<int> fut = tsk.get_future();
    tsk(1979);
    int r_value = fut.get();
    cout << "Output: " << r_value << endl;

    return 0;
}
```

Future providers

- Example: *using packaged_task<> in multithreading*

```
#include <future>
#include <thread>
...
using namespace std;
int compute_double(int value) { return value*2; }

int main() {
     packaged_task<int(int)> tsk(compute_double);
     future<int> fut = tsk.get_future();
    thread th(std::move(tsk), 1979);
    int r_value = fut.get();
    cout << "Output: " << r_value << endl;
    th.join();
    return 0;
}
```

Task-based vs thread-based approaches

- *Task-based* approaches
  - Functions return value accessible
  - Smart task/thread spawning with default policy
    - CPU load balancing → the C++ library can run the function without spawning a thread
    - Avoid the raising of `std::system_error` in case of thread number reachead the system limit
  - *Future* objects allows us to catch exceptions thrown by the function
    - While with `std::thread()` the program terminates

- *Thread-based* approaches
  - Used to execute tasks that do not terminate till the end of the application
    - A thread entry point function is like a second, concurrent main()
  - More general concurrency model, can be used for *thread-based design patterns*
  - Allows us to access to the *pthread* native handle
    - Useful for advanced management (priority, affinity, scheduling policies,...)

Outline

Introduction

- Multi-tasking implementations

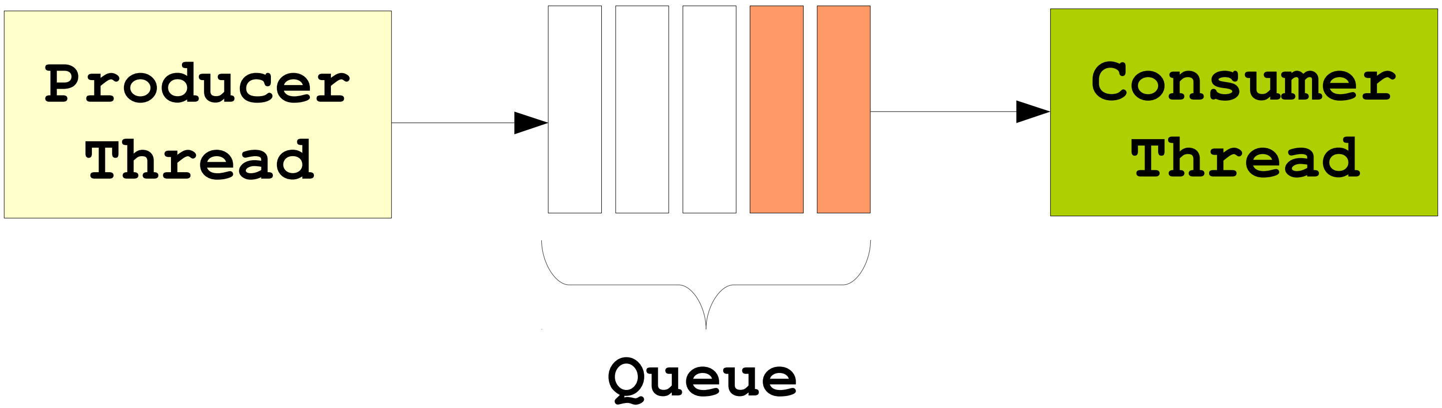
C++11 threading support

- Thread creation
- Synchronization
- Mutual exclusion and pitfalls
- Condition variables
- Task-based approaches

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();
    size_t size();
private:
    SynchronizedQueue(const SynchronizedQueue &)=delete;
    SynchronizedQueue & operator=(const SynchronizedQueue &)=delete;
    std::list<T> queue;
    std::mutex myMutex;
    std::condition_variable 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;
}
size_t SynchronizedQueue::size() {
     std::unique_lock<std::mutex> lck(myMutex);
     return queue.size();
}
#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 &)=delete;
    SynchronizedQueue & operator=(const SynchronizedQueue &)=delete;
    std::list<T> queue;
    std::mutex myMutex;
//  std::condition_variable 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();
    virtual ~ActiveObject();
private:
    virtual void run();
    ActiveObject(const ActiveObject &)=delete;
    ActiveObject& operator=(const ActiveObject &)=delete;
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(&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

- We can use it to implement threads communicating through a producer/consumer approach

- In the provided 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

C++11 constructs

`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
- Tasks 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));
    …
}
```

Reactor limitations

- 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 processors we can have multiple executor threads picking tasks from the same queue
  - We need a different pattern for this

ThreadPool

- One (or more) queue(s) of tasks/jobs and a fixed set of *worker threads*
- Better control over thread creation overhead
- Some design issues to consider...
  - How many worker threads to use?
    - A number somehow related to the number of available CPU cores
  - How to allocate tasks to threads?
  - Wait for a task to complete or not?

ThreadPool (version 1)

- threadpool.h (1/2)

```
#ifndef THREADPOOL_H
#define THREADPOOL_H
#include <atomic>
#include <functional>
#include <list>
#include <mutex>
#include <thread>
#include <vector>

class ThreadPool
{
    private:
        std::atomic<bool> done;                 //! Thread pool status
        unsigned int thread_count;              //! Thread pool size
        std::mutex wq_mutex;
        std::list<std::function<void()>> work_queue;
        std::vector<std::thread> threads;       //! Worker threads
        void worker_thread();
…
```

ThreadPool (version 1)

- threadpool.h (2/2)

```
    public:
        ThreadPool(int nr_threads = 0);

        virtual ~ThreadPool();

        void pushTask(std::function<void ()> func) {
            std::unique_lock<std::mutex> lck(wq_mutex);
            work_queue.push_back(std::function<void()>(func));
        }

        void getWorkQueueLength() {
            std::unique_lock<std::mutex> lck(wq_mutex);
            return work_queue.size();
        }
};

#endif // THREADPOOL_H
```

ThreadPool (version 1)

- threadpool.cpp (1/2)

```
#include "threadpool.h"

ThreadPool::ThreadPool(int nr_threads): done(false) {
    if (nr_threads <= 0)
        thread_count = std::thread::hardware_concurrency();
    else
        thread_count = nr_threads;
    for (unsigned int i=0; i < thread_count; ++i)
        threads.push_back(
            std::thread(&ThreadPool::worker_thread, this));
}

ThreadPool::~ThreadPool() {
    done = true;
    for (auto & th: threads)
        if (th.joinable())
            th.join();
}
```

ThreadPool (version 1)

- threadpool.cpp (2/2)

```
void ThreadPool::worker_thread()
{
    while (!done) {
        wq_mutex.lock();
        if (work_queue.empty()) {
            wq_mutex.unlock();
            std::this_thread::yield();
        }
        else {
            std::function<void()> task = work_queue.front();
            work_queue.pop_front();
            wq_mutex.unlock();
            task();  // Run the function/job
        }
    }
}
```

Design patterns

ThreadPool

- A given number of worker threads is spawned when creating the ThreadPool object (**`nr_threads`**)
  - **`nr_threads`** = #CPUs if not specified
- A shared work queue includes functions (jobs) to execute (**`work_queue`**)
- Each worker thread check for the presence of some work
  - If there is some take it from the queue and execute the function
  - If the work queue is empty let the OS scheduler to execute other threads (**`std::thread::yield`**)
- When the ThreadPool object is destroyed…
  - The temination condition is set (**`done = true`**), allowing the worker threads to exit from the loop
  - All the worker threads are joined

ThreadPool

- This implementation has a couple of big issues
  - The call to `std::thread::yield` causes the idle threads to continuously spin, consuming CPU cycles without doing any useful work
  - We can improve the implementation by reusing a class, already introduced, to manage the work queue (SynchronizedQueue)
    - The class is already thread-safe, so we can remove the synchronization statement used to access the work queue

ThreadPool (version 2)

- thread_pool2.h (1/2)

```
#ifndef THREADPOOL_H
#define THREADPOOL_H
#include <atomic>
#include <functional>
#include <mutex>
#include <thread>
#include <vector>
#include “synchronized_queue.h”

class ThreadPool
{
    private:
        std::atomic<bool> done;                 //! Thread pool status
        unsigned int thread_count;              //! Thread pool size
        SynchronizedQueue<std::function<void()>> work_queue;
        std::vector<std::thread> threads;       //! Worker threads
        void worker_thread();
…
```

ThreadPool (version 2)

- thread_pool2.h (2/2)

```
    public:
        ThreadPool(int nr_threads = 0);

        virtual ~ThreadPool();

        void pushTask(std::function<void ()> func) {
            // SynchronizedQueue guarantees mutual exclusive access
            work_queue.put(func);

        }

        void getWorkQueueLength() {
            return work_queue.size();
        }
};

#endif // THREADPOOL_H
```

ThreadPool (version 2)

- thread_pool2.cpp (1/2)

```
#include "threadpool.h"
void doNothing() {}

ThreadPool::ThreadPool(int nr_threads): done(false) {
    if (nr_threads <= 0)
        thread_count = std::thread::hardware_concurrency();
    else
        thread_count = nr_threads;
    for (unsigned int i=0; i < thread_count; ++i)
        threads.push_back(
            std::thread(&ThreadPool::worker_thread, this));
}

ThreadPool::~ThreadPool() {
    done = true;
    for (unsigned int i=0; i < thread_count; ++i) pushTask(&doNothing);
    for (auto & th: threads)
        if (th.joinable())
            th.join();
}
```

ThreadPool (version 2)

- thread_pool2.cpp (2/2)

```
void ThreadPool::worker_thread()
{
    while (!done) {
        work_queue.get()(); //Get a function and call it
    }
}
```

- Much simpler implementation
  - Tasks/jobs are picked from the SynchronizedQueue and executed
- The idle threads do no spin anymore, wasting CPU cycles
  - They block on the SynchronizedQueue condition variable
  - For this we need to push “doNothing()” tasks while executing the class destructor to wake up all the worker threads and successfully join them

Sequential and Parallel implementations

Comparison

- Example: *Fibonacci sequential implementation*

```
// Return the n'th Fibonacci number
unsigned long fibonacci(unsigned int n) {
    if (n == 0)
        return 0;

    unsigned long prev = 0;
    unsigned long curr = 1;
    for (unsigned int i = 1; i < n; ++i) {
        unsigned long next = prev + curr;
        prev = curr;
        curr = next;
    }
    return curr;
}
```

Comparison

- Example: *Fibonacci parallel implementation*

```
...
void parallelExecutor(const list<function<void ()>>& tasks) {
    list<shared_ptr<thread>> threads;
    for(auto f : tasks)
        threads.push_back(make_shared<thread>(f));
}
```

```
struct Result
{
    mutex m;
    int r;
};
```

```
// Return the n'th Fibonacci number
void fibonacci(int n, Result *result) {
    unique_lock<mutex> lock(result->m);
    if(n <= 1) {
        result->r = 1;
    } else {
        Result result2;
        ParallelExecutor(
            { bind(fibonacci,n-1,result), bind(fibonacci,n-2,&result2) });
        unique_lock<mutex> lock1(result->m);
        unique_lock<mutex> lock2(result2.m);
        result->r += result2.r;
    }
}
```

Comparison

- Which version do you expect to run faster?
  - Parallel implementation is likely to be much worse than a sequential one
  - For each iteration of the fibonacci algorithm a thread is created, which is an expensive operation
    - System call to the OS, the allocation of a stack, new thread in the scheduler data structures, context switches
    - All these operations are justified only if threads perform significant work
  - The sequential version would only...
    - Allocate a certain number of stack frames, in case of recursive implementation
    - Loop to execute an addition and an assignment operation