

# Multithreading in C++ and C#

Máté Cserép

*Eötvös Loránd University, Faculty of Informatics*

*May 2021, Budapest*

# Parallel computing

- Computers can carry out multiple tasks parallelly.
- Parallel computing is often a requirement for even simple applications
  - e.g. a basic word processor should handle user input regardless whether it is being busy with updating the user interface, semantic analysis, etc.
- C++ and C# have multiple tools to support parallel programming.

# Parallel computing

## Process

- Contains a complete execution environment and runtime resources, like memory.
- By default our C++/C# program runs as a single process.

## Thread

- A process may contain multiple threads.
- Which share virtual address space and system resources.
- Threads are more lightweight compared to processes.
- Each process starts with an initial thread, often called primary or main thread.

**We will focus on multi thread programming.**

# Multithreading in C++

*// Multithreading is just one damn thing after, before, or simultaneous with another.*

Andrei Alexandrescu

# Multithreading in C++

Beyond the errors which can occur in single-threaded programs, multithreaded environments are subject to additional errors:

- Race conditions
- Deadlock, livelock
- Priority failures (priority inversion, starvation, etc.)

Moreover testing and debugging of multithreaded programs are harder. Multithreaded programs are *non-deterministic*. Failures are often *non-repeatable*. Debugged code can produce very different results than non-debugged ones. Testing on single processor hardware may produce different results than testing on multiprocessor hardware.

# Atomicity

In C++11 this is undefined behaviour, in C++98/03 not even that.

```
int x, y;  
  
// thread 1  
x = 1;  
y = 2;
```

```
// thread 2  
std::cout << y << ", ";  
std::cout << x << std::endl;
```

## Workaround with mutexes:

```
int x, y;  
std::mutex x_mutex, y_mutex;  
  
// thread 1  
x_mutex.lock();  
x = 1;  
x_mutex.unlock();  
y_mutex.lock();  
y = 2;  
y_mutex.unlock();
```

```
// thread 2  
y_mutex.lock();  
std::cout << y << ", ";  
y_mutex.unlock();  
x_mutex.lock();  
std::cout << x << std::endl;  
x_mutex.unlock();
```

## Workaround with atomic:

```
std::atomic<int> x, y;  
  
// thread 1  
x.store(1);  
y.store(2);
```

```
// thread 2  
std::cout << y.load() << ", ";  
std::cout << x.load() << std::endl;
```

# Threads

```
namespace std
{
    class thread
    {
    public:
        typedef native_handle /* ... */;
        typedef id /* ... */;

        thread() noexcept;           // does not represent a thread
        thread(thread&& other) noexcept; // move constructor
        ~thread();                   // if joinable() calls std::terminate()

        template <typename Function, typename... Args> // copies args to thread local
        explicit thread(Function&& f, Arg&&... args); // then execute f with args

        thread(const thread&) = delete;           // no copy
        thread& operator=(thread&& other) noexcept; // move
        void swap(thread& other);                 // swap

        bool joinable() const; // thread object owns a physical thread
        void join();           // blocks current thread until *this finish
        void detach();        // separates physical thread from the thread object

        std::thread::id get_id() const;           // std::this_thread
        static unsigned int hardware_concurrency(); // supported concurrent threads
        native_handle_type native_handle();       // e.g. thread id
    };
}
```

# Typesafe parameter passing to the thread

Creates a new thread of execution with  $t$ , which calls  $f(3, \text{"hello"})$ , where arguments are copied (as is) into an internal storage (even if the function takes them as reference).

```
void f(int i, const std::string& s);  
std::thread t(f, 3, "Hello");
```

If an exception occurs, it will be thrown in the hosting thread.

$f$  can be any callable function, e.g. *operator()*:

```
class f  
{  
public:  
    f(int i = 0, std::string s = "") : _i(i), _s(s) { }  
    void operator()() const  
    {  
        // background activity  
    }  
    int _i;  
    std::string _s;  
};  
  
std::thread t(f()); // Most vexing parse (Scott Meyers: Effective STL)  
std::thread t((f(3, "Hello"))); // OK
```



# Passing parameter by reference

By default all arguments are copied by value, even if the function takes them as reference.

```
void f(int i, const std::string&)
{
    std::cout << "Hello Concurrent World!" << std::endl;
}

int main()
{
    int i = 3;
    std::string s("Hello");

    // Will copy both i and s
    //std::thread t(f, i, s);

    // We can prevent the copy by using reference wrapper
    std::thread t(f, std::ref(i), std::ref(s));

    // If the thread destructor runs and the thread is joinable, then
    // std::system_error will be thrown.

    // Use join() or detach() to avoid that.
    t.join();
    return 0;
}
```

# Alternative destructor strategies

*Scott Meyers*

- Implicit join: the destructor waits until the thread execution is completed. Hard to detect performance issues.
- Implicit detach: destructor runs, but the underlying thread will still run. Destructor may free memory but the thread may try to access them.

# Possible problems

Still, there is possible to make wrong code (of course, this is C++).  
Better to avoid pointers and references, or *join()*.

```
struct func
{
    int& i;
    func(int& i_) : i (i_) { }

    void operator()()
    {
        for(unsigned int j = 0; j < 1000000; ++j)
        {
            do_something(i); // i may refer to a destroyed variable
        }
    }
};

int main()
{
    int some_local_state = 0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach(); // don't wait the thread to finish
    return 0;
} // some_local_state is destroyed, but the thread is likely still running.
```

# Scoped thread

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_): t(std::move(t_))
    {
        if(!t.joinable())
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func;

void f()
{
    int some_local_state;
    scoped_thread t(std::thread(func(some_local_state)));
    do_something_in_current_thread();
}
```

Source: *C++ Concurrency In Action*, by Anthony Williams

Implementation can also be found in the Boost Library

`<boost/thread/scoped_thread.hpp>`

# Possible problems

```
void f(int i, const std::string& s);

std::thread t(f, 3, "Hello");
```

*"Hello"* is passed to *f* as *const char \** and converted to *std::string* in the new thread. This can lead to problems, e.g.:

```
void f(int i, const std::string& s);

void bad(int some_param)
{
    char buffer[1024];
    sprintf(buffer, "%i", some_param);
    std::thread t(f, 3, buffer);
    t.detach();
}

void good(int some_param)
{
    char buffer[1024];
    sprintf(buffer, "%i", some_param);
    std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

# Threads with STL

*sth::thread* is compatible with the STL containers.

```
void do_work(unsigned id);

int main()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.push_back(std::thread(do_work,i));
    }

    std::for_each(threads.begin(), threads.end(),
                 [](std::thread& t) { t.join(); }); // join all threads

    std::for_each(threads.begin(), threads.end(),      // alternative:
                 std::mem_fn(&std::thread::join)); // generates functor for function

    return 0;
}
```

# Synchronization objects

## Mutex:

```
std::mutex m;  
int sh; // shared data  
  
void f()  
{  
    /* ... */  
    m.lock();  
    // manipulate shared data:  
    sh += 1;  
    m.unlock();  
    /* ... */  
}
```

## Recursive mutex:

```
std::recursive_mutex m;  
int sh; // shared data  
  
void f(int i)  
{  
    /* ... */  
    m.lock();  
    // manipulate shared data:  
    sh += 1;  
    if (--i > 0) f(i);  
    m.unlock();  
    /* ... */  
}
```

# Synchronization objects

## Timed mutex:

```
std::timed_mutex m;
int sh; // shared data

void f()
{
    /* ... */
    if (m.try_lock_for(std::chrono::seconds(10))) {
        // we got the mutex, manipulate shared data:
        sh += 1;
        m.unlock();
    }
    else {
        // we didn't get the mutex; do something else
    }
}

void g()
{
    /* ... */
    if (m.try_lock_until(midnight)) {
        // we got the mutex, manipulate shared data:
        sh+=1;
        m.unlock();
    }
    else {
        // we didn't get the mutex; do something else
    }
}
```



# Locks

Locks support the Resource Allocation Is Initialization (RAII) idiom.

```
std::list<int> l;  
std::mutex    m;  
  
void add_to_list(int value);  
{  
    // lock acquired - with RAII style lock management  
    std::lock_guard<std::mutex> guard(m);  
    l.push_back(value);  
} // lock released
```

**Pointers or references pointing out from  
the guarded area can be an issue!**

# Deadlocks

The code below can result in a deadlock when  $a < b$  and  $b < a$  are simultaneously evaluated on 2 threads.

```
template <class T>
bool operator<(const T& lhs, const X& rhs)
{
    if (&lhs == &rhs)
        return false;

    lhs.m.lock(); rhs.m.lock();
    bool result = lhs.data < rhs.data;
    lhs.m.unlock(); rhs.m.unlock();
    return result;
}
```

## Avoid deadlocks

1. Avoid nested locks
2. Avoid user defined call when holding a lock
3. Acquire locks in a fixed order

# Deadlocks

A correct solution to avoid deadlock:

```
template <class T>
bool operator<(T const& lhs, X const& rhs)
{
    if (&lhs == &rhs)
        return false;

    // std::lock - locks two or more mutexes
    std::lock(lhs.m, rhs.m);

    // std::adopt_lock - assume the calling thread already has ownership
    std::lock_guard<std::mutex> lock_lhs(lhs.m, std::adopt_lock);
    std::lock_guard<std::mutex> lock_rhs(rhs.m, std::adopt_lock);

    return lhs.data < rhs.data;
}
```

With the lock guards, mutexes are released with RAII.

# Deadlocks

Another correct solution with different approach:

```
template <class T>
bool operator<(T const& lhs, X const& rhs)
{
    if (&lhs == &rhs)
        return false;

    // std::unique_locks constructed with defer_lock can be locked
    // manually, by using lock() on the lock object ...
    std::unique_lock<std::mutex> lock_lhs(lhs.m, std::defer_lock);
    std::unique_lock<std::mutex> lock_rhs(rhs.m, std::defer_lock);
    // lock_lhs.owns_lock() now false

    // ... or passing to std::lock
    std::lock(lock_lhs, lock_rhs); // designed to avoid dead-lock
    // also there is an unlock() member function

    // lock_lhs.owns_lock() now true
    return lhs.data < rhs.data;
}
```

*std::unique\_lock* can be locked and unlocked.

(It is also moveable, but not copyable, but that is not a factor here.)

# Deadlocks

C++17 simplifies the problem with the introduction of *scoped\_lock*, specifically designed for locking (and releasing) multiple mutexes and the same time in RAII style:

```
template <class T>
bool operator<(T const& lhs, X const& rhs)
{
    if (&lhs == &rhs )
        return false;

    // designed to avoid dead-lock
    std::scoped_lock lock(lhs.m, rhs.m);

    return lhs.data < rhs.data;
}
```

# Singleton Pattern: naïve

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        std::unique_lock<std::mutex> lock(resource_mutex);
        if (!resource_ptr)
            resource_ptr.reset(new T(/* ... */));
        lock.unlock();
        return resource_ptr;
    }
private:
    std::shared_ptr<T> resource_ptr;
    mutable std::mutex resource_mutex;
};
```

Problem: while the problematic race condition is connected only to the initialization of the Singleton instance, the critical section is executed for every calls of the *instance()* method. Such an excessive usage of the locking mechanism may cause serious overhead which could not be acceptable.

# Singleton Pattern: DCLP

## Double-Checked Locking Pattern

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        if (!resource_ptr) // 1
        {
            std::unique_lock<std::mutex> lock(resource_mutex);
            if (!resource_ptr)
                resource_ptr.reset(new T(/* ... */)); // 2
            lock.unlock();
        }
        return resource_ptr;
    }
private:
    std::shared_ptr<T> resource_ptr;
    mutable std::mutex resource_mutex;
};
```

Problem: load in (1) and store in (2) is not synchronized.

This can lead to a bug with non-atomic pointer or integral assignment semantics; or if an overly-aggressive compiler optimizes *resource\_ptr* (e.g. storing it in a register).

# Singleton Pattern: *call\_once*

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        std::call_once(resource_init_flag, init_resource);
        return resource_ptr;
    }
private:
    void init_resource()
    {
        resource_ptr.reset(new T(/* ... */));
    }
    std::shared_ptr<T> resource_ptr;
    std::once          resource_init_flag; // can't be moved or copied
};
```

*std::call\_once* is guaranteed to execute its callable parameter exactly once, even if called from several threads.



# Singleton Pattern: Meyers singleton

```
class MySingleton;  
MySingleton& MySingletonInstance()  
{  
    static MySingleton _instance;  
    return _instance;  
}
```

C++11 guarantees that this is thread safe!

# Condition variable

## Classical producer-consumer example:

```
std::mutex                my_mutex;
std::queue<data_t>        my_queue;
std::conditional_variable data_cond; // conditional variable

void producer() {
    while (more_data_to_produce())
    {
        const data_t data = produce_data();
        std::lock_guard<std::mutex> prod_lock(my_mutex); // guard the push
        my_queue.push(data);
        data_cond.notify_one(); // notify the waiting thread to evaluate cond.
    }
}

void consumer() {
    while (true)
    {
        std::unique_lock<std::mutex> cons_lock(my_mutex); // not lock_guard
        data_cond.wait(cons_lock,                          // returns if lambda returns true
            [&my_queue]{return !my_queue.empty();});     // else unlocks and waits
        data_t data = my_queue.front();                   // lock is hold here to protect pop...
        my_queue.pop();
        cons_lock.unlock();                               // ... until here
        consume_data(data);
    }
}
```

# Condition variable

- During the wait the condition variable may check the condition any time, but under the protection of the *mutex* and returns immediately if condition is true.
- Spurious wake: wake up without notification from other thread. Undefined times and frequency, so it is better to avoid functions with side effect. (E.g. using a counter in lambda to check how many notifications were is typically bad.)

# Futures and Promises

1. Future is a read-only placeholder view of a variable.
2. Promise is a writable, single assignment container (set the value of future).
3. Futures are results of asynchronous function calls. When I execute that function I won't get the result, but get a *future* which will hold the result when the function completed.
4. A future is also capable to store exceptions.
5. With shared futures multiple threads can wait for a single shared async result.

# Futures

```
int f(int);
void do_other_stuff();

int main()
{
    std::future<int> the_answer = std::async(f, 1);
    do_other_stuff();
    std::cout<< "The answer is " << the_answer.get() << std::endl;
    return 0;
}
```

The `std::async()` executes the task either in a new thread or on `get()`.

```
// starts in a new thread
auto fut1 = std::async(std::launch::async, f, 1);

// run in the same thread on wait() or get()
auto fut2 = std::async(std::launch::deferred, f, 2);

// default: implementation chooses
auto fut3 = std::async(std::launch::deferred | std::launch::async, f, 3);

// default: implementation chooses
auto fut4 = std::async(f, 4);
```

If no `wait()` or `get()` is called, then the task may not be executed at all.

# Futures and exceptions

```
double square_root(double x)
{
    if (x < 0)
    {
        throw std::out_of_range("x is negative");
    }
    return sqrt(x);
}

int main()
{
    std::future<double> fut = std::async(square_root, -1);
    double res = fut.get(); // f becomes ready on exception and rethrows
    return 0;
}
```

## Further future methods

- *fut.valid()*: future has a shared state
- *fut.wait()*: wait until result is available
- *fut.wait\_for()*: timeout duration
- *fut.wait\_until()*: wait until specific time point

# Futures' destructors

Keep in mind that the futures has a special shared state, which demands that `future::~~future` blocks.

```
double long_calculation(int n)
{
    /* ... */
}

int main()
{
    std::async(std::launch::async, long_calculation, 42); // ~future blocks
    std::async(std::launch::async, long_calculation, 100); // ~future blocks
}
```

For real asynchronous you need to keep the returned future:

```
int main()
{
    std::future<double> fut1 = std::async(std::launch::async, long_calculation, 42);
    // no blocking
    std::future<double> fut2 = std::async(std::launch::async, long_calculation, 100);
    // no blocking
}
```

# Promises

A promise is a tool for passing the return value (or exception) from a thread executing a function to the thread that consumes the result using future.

```
void asyncFun(std::promise<int> myPromise)
{
    int result;
    try
    {
        // calculate the result
        myPromise.set_value(result);
    }
    catch (...)
    {
        myPromise.set_exception(std::current_exception());
    }
}
```



# Promises

```
void asyncFun(std::promise<int> myPromise)
{
    int result;
    try
    {
        // calculate the result
        myPromise.set_value(result);
    }
    catch (...)
    {
        myPromise.set_exception(std::current_exception());
    }
}
```

```
int main()
{
    std::promise<int> intPromise;
    std::future<int> intFuture = intPromise.getFuture();
    std::thread t(asyncFun, std::move(intPromise));

    // do other stuff here, while asyncFun is working

    int result = intFuture.get(); // may throw exception
    return 0;
}
```

# Packaged task

A higher level tool than promises.

```
double square_root(double x)
{
    if ( x < 0 )
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}

int main()
{
    double x = 4.0;

    std::packaged_task<double(double)> tsk(square_root);
    std::future<double> fut = tsk.get_future(); // future will be ready when task completes

    std::thread t(std::move(tsk), x); // make sure, task starts immediately
                                     // on different thread
                                     // thread can be joined, detached

    double res = fut.get(); // using the future
    return 0;
}
```

# Packaged task implementation

```
template <typename> class my_task;

template <typename R, typename ...Args>
class my_task<R(Args...)>
{
    std::function<R(Args...)> fn;
    std::promise<R> pr;
public:
    template <typename ...Ts>
    explicit my_task(Ts&&... ts) : fn(std::forward<Ts>(ts)...) { }

    template <typename ...Ts>
    void operator()(Ts&&... ts)
    {
        pr.set_value(fn(std::forward<Ts>(ts)...));
    }

    std::future<R> get_future() { return pr.get_future(); }
    // disable copy, default move
};
```

# Packaged task vs async

In the end a *std::packaged\_task* is just a lower level feature for implementing *std::async* (which is why it can do more than *std::async* if used together with other lower level stuff, like *std::thread*).

Simply spoken a *std::packaged\_task* is a *std::function* linked to a *std::future* and *std::async* wraps and calls a *std::packaged\_task* (possibly in a different thread).

# Parallel STL (C++17)

C++17 brings us parallel algorithms, so the well known STL algorithms (*std::find\_if*, *std::for\_each*, *std::sort*, etc.) get a support for parallel (or *vectorized*) execution.

```
vector<int> v = { /* ... */ };

// standard sequential sort
std::sort(v.begin(), v.end());

// sequential execution
std::sort(std::parallel::seq, v.begin(), v.end());

// permitting parallel execution
std::sort(std::parallel::par, v.begin(), v.end());

// permitting vectorized execution (only since C++20)
std::sort(std::parallel::unseq, v.begin(), v.end());

// permitting parallel and vectorized execution
std::sort(std::parallel::par_unseq, v.begin(), v.end());
```

# Parallel STL (C++17)

## What is *vectorized* (or *unsequenced*) execution?

- Most modern CPUs can execute SIMD (*single instruction, multiple data*) operations, significantly boosting efficiency.
- A SIMD operation is where the same instruction is executed on multiple data. SIMD is data level parallelism without thread concurrency.
- With parallel and vectorized execution policies, it is the programmer's responsibility to avoid data races and deadlocks.

# Parallel STL (C++17)

Parallel STL is only implemented in modern compilers, so keep in mind where you can use this new feature.

- GNU GCC: supported since version 9.1 (May 2019)
- Clang: no support (soon thanks to Intel's "donation")
- MSVC: supported since MSVC 19.14  
(Visual Studio 2017 with at least version 15.7)

Support is not necessarily complete, e.g. MSVC does not implement the vectorized execution policy.

Follow current state at:

[https://en.cppreference.com/w/cpp/compiler\\_support](https://en.cppreference.com/w/cpp/compiler_support)

Look for *"Standardization of Parallelism TS"*.

# Multithreading in C#

*“ Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren't possible, and discovers that they didn't actually understand it yet after all.*

Herb Sutter

Chair of the ISO C++ standards committee, Microsoft



# Atomicity

Atomic data types: *bool, char, byte, sbyte, short, ushort, uint, int, float*, and reference types.

Non-atomic data types: *long, ulong, double, decimal*, etc.

There is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

Basic atomicity can be achieved through the methods of the *Interlocked* class:

```
class SomeType { /* ... */ }

public static Program {
    public static void Main(string[] args) {

        int x = 41;
        Interlocked.Increment(ref x);    // increment x

        SomeType y = new SomeType();
        SomeType z = new SomeType();
        // ...
        Interlocked.Exchange(ref y, z); // replace y with z
    }
}
```

# Blocking: Mutexes

```
public Stack<T>
{
    private Mutex mutex;
    private IList<T> values;

    public Stack()
    {
        mutex = new Mutex();
        values = new List<T>();
    }

    public void Push(T item);
    {
        mutex.WaitOne();
        values.Add(item); // critical section
        mutex.ReleaseMutex();
    }
}
```

Can also wait until a timeout reached:  
*mutex.WaitOne(Int32)* and *mutex.WaitOne(TimeSpan)*

# Blocking: Semaphore

```
public Stack<T>
{
    private Semaphore sem;
    private IList<T> values;

    public Stack()
    {
        sem = new Semaphore();
        values = new List<T>();
    }

    public void Push(T item);
    {
        sem.WaitOne();
        values.Add(item); // critical section
        sem.Release();
    }
}
```

Can specify the number of initial entries (ownership) and the maximum number of concurrent entries:

```
Semaphore sem = new Semaphore(0, 3);
```

# Blocking: Monitors

```
public Stack<T>
{
    private IList<T> values;

    public Stack()
    {
        values = new List<T>();
    }

    public void Push(T item);
    {
        Monitor.Enter(values);
        values.Add(item); // critical section
        Monitor.Exit(values);
    }
}
```

Same as using the *lock* statement:

```
public void Push(T item);
{
    lock(values)
    {
        values.Add(item); // critical section
    }
}
```

# Mutexes vs. Semaphores vs. Monitors

## **Mutex:**

- can be named
- scope is system-wide
- good for synchronising between different processes (applications)

## **Semaphore:**

- can be named
- more lightweight
- maximum scope is application-wide
- good for synchronising between threads

## **Monitor:**

- unnamed
- scope is the same the object it locks on
  - maximum scope is application-wide

# Concurrent collections

Thread-safe, mutually exclusive collections are part of the .NET Standard Library, under the *System.Collections.Concurrent* namespace

- *ConcurrentBag, ConcurrentDictionary, ConcurrentQueue, ConcurrentStack, BlockingCollection* (producer-consumer)
- The signature of their operations are a little different, but they also inherit the usual interfaces, e.g.:

```
IDictionary<String, Object> dictionary =  
    new ConcurrentDictionary<String, Object>();
```

# Threads

```
class Program {
    public static void DoWork() {
        Console.WriteLine("Child thread starts");

        Console.WriteLine("Child thread goes to sleep");
        Thread.Sleep(5000); // the thread is paused for 5000 milliseconds
        Console.WriteLine("Child thread resumes and finishes");
    }

    static void Main(string[] args) {
        ThreadStart childJob = new ThreadStart(DoWork);
        Console.WriteLine("Main thread starts");

        Thread childThread = new Thread(childJob);
        childThread.Start();

        Console.WriteLine("Main thread waiting");
        childThread.Join();
        Console.WriteLine("Main thread finishes");
    }
}
```

[See code example](#)

# Threads: passing parameters

```
class Program {
    public static void DoWork(object obj) {
        Console.WriteLine("Child thread starts");

        if (obj is String)
            Console.WriteLine(obj as String);
        else
            throw new ArgumentException("Parameter is not a string.", nameof(obj));

        Console.WriteLine("Child thread goes to sleep");
        Thread.Sleep(5000); // the thread is paused for 5000 milliseconds
        Console.WriteLine("Child thread resumes and finishes");
    }

    static void Main(string[] args) {
        ParameterizedThreadStart childJob = new ParameterizedThreadStart(DoWork);
        Console.WriteLine("Main thread starts");

        Thread childThread = new Thread(childJob);
        childThread.Start("Message from Main");

        Console.WriteLine("Main thread waiting");
        childThread.Join();
        Console.WriteLine("Main thread finishes");
    }
}
```



# Threads

Problems with plain *Thread* objects:

- cannot pass typed parameters  
(shared data members can be used)
- cannot return result  
(shared data members can be used)
- no exception forwarding from child thread to main thread

# Tasks (since .NET 4.0)

- Higher abstraction level solution for asynchronous or delayed computation (compared to Thread)
- Tasks execute an operation given as a lambda-expression (*Action, Func*).
- Tasks can be executed by a single method call (*Task.Run*) or can be instantiated and executed later (*Start*). The factory design pattern can also be utilized (*Task.Factory.StartNew*).
- The result of a Task can be retrieved through the *Result* property (will wait to be accessible).

# Tasks

```
private Int32 Compute(){ /* ... */ }
    // calculation which produces a result

private void RunCompute() {

    Int32 result = Task.Run(() => Compute()).Result;
        // execute task and wait for the result

    // ...
}
```

```
private Int32 Compute(){ /* ... */ }
    // calculation which produces a result

private void RunCompute() {
    Task<Int32> myTask = new Task<Int32>(() => Compute());
        // create a task with the job given
    myTask.Start(); // start the task

    // ...

    Int32 result = myTask.Result;
        // wait for the result

    // ...
}
```

# Tasks

```
class Program {
    public static int Add(int a, int b) {
        Console.WriteLine("Child thread starts");
        int result = a + b;

        Console.WriteLine("Child thread goes to sleep");
        Thread.Sleep(5000); // the thread is paused for 5000 milliseconds

        Console.WriteLine("Child thread resumes and finishes");
        return result;
    }

    public static void Main(string[] args) {
        int x = 30;
        int y = 12;

        Task<int> task = new Task<int>(() => Add(x, y));
        Console.WriteLine("Main thread starts");
        task.Start();

        Console.WriteLine("Main thread waiting");
        int sum = task.Result; // blocks until result is ready
                               // alternative: task.Wait() and its overloads
        Console.WriteLine("Main thread finishes, sum = {0}", sum);
    }
}
```

# Tasks: exception handling

Unhandled exceptions thrown by user code that is running inside a task are propagated back to the calling thread.

Multiple exception can be thrown (e.g. when on waiting multiple child tasks), so the *Task* infrastructure wraps them in an *AggregateException* instance.

```
public static void Main(string[] args) {
    Console.WriteLine("Main thread starts");
    Task<int> taskA = DoWorkAsync(42);
    Task<int> taskB = DoWorkAsync(100);

    Console.WriteLine("Main thread waiting");
    try {
        Task.WaitAll(new Task[] { taskA, taskB });
        // taskA.Result and taskB.Result are available at this point
    }
    catch (AggregateException ae) {
        foreach (var e in ae.InnerExceptions) {
            // handle exception ...
        }
    }
    Console.WriteLine("Main thread finishes");
}
```

# Tasks: async / await

```
class Program {
    public static int Add(int a, int b) {
        /* ... */
    }

    public static async Task<int> AddAsync(int a, int b)
    {
        return await Task.Run(() => Add(a, b));
    }

    public static void Main(string[] args) {
        int x = 30;
        int y = 12;

        Console.WriteLine("Main thread starts");
        Task<int> task = AddAsync(x, y);

        Console.WriteLine("Main thread waiting");
        int sum = task.Result;
        Console.WriteLine("Main thread finishes, sum = {0}", sum);
    }
}
```

[See code example](#)

# Tasks: async / await

Since .NET 4.5 methods of standard library which should be run as a background tasks are available as asynchronous operations.

- By convention these methods has the Async suffix in their name.
- E.g. I/O operations can be slow (compared to CPU and memory operations):

```
StreamReader reader = new StreamReader("somefile.txt");  
String firstLine = await reader.ReadLineAsync();
```

# Tasks: cancellation

- Tasks can be gracefully interrupted through a *CancellationToken*.
- The token can be fetched from a *CancellationTokenSource* and the interruption can be achieved by the *Cancel* method.
- This does not abort the task, but we can programmatically check the *IsCancellationRequested* property and cancel the task.

Threads can be terminated through the *Abort* method unconditionally, which is considered an obsolete solution.



# Tasks: cancellation

```
class Program {  
  
    public static void Main(string[] args) {  
        // ...  
  
        CancellationTokenSource source = new CancellationTokenSource(); // token source  
        CancellationToken token = source.Token; // token  
  
        Task.Run(() => {  
            // ...  
            if (token.IsCancellationRequested)  
                // if requested  
                return; // we cancel the execution  
            // ...  
        }, token); // pass the cancellation token  
  
        // ...  
    }  
}
```

# Tasks: synchronization

- Tasks can be synchronized using the *TaskScheduler* type.
- Tasks can be initialized with a *TaskScheduler* param.
- The static *FromCurrentSynchronizationContext* method provides an easy solution to synchronize into the current thread.
- Usually we do not want to synchronize the executed operation, just to access the UI elements.
  - We can execute an outer, asynchronous task.
  - Inside it we can run a synchronized task.
  - After the outer task finished, we can chain further operations with the *ContinueWith* method.

# Tasks: synchronization

```
class Program {  
  
    public static void Main(string[] args) {  
        // ...  
  
        TaskScheduler scheduler = TaskScheduler.FromCurrentSynchronizationContext();  
        // scheduler for synchronization  
  
        Task.Factory.StartNew(() => { ... }, ..., ..., scheduler)  
        // the task will be executed synchronously  
  
        Task.Factory.StartNew(() => { ... })  
        .ContinueWith(() => { label.Text = "Ready." }, scheduler);  
        // the task is executed asynchronously,  
        // then executes a synchronous operation  
        // to provide a thread-safe way to access the UI  
  
        // ...  
    }  
}
```