Advanced Development Techniques

Parallel programming

Sipos Miklós

Óbudai Egyetem Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet
2021

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Contents

Processes and threads II.

- Task

Task

- Asynchronously finished “job”
- The ThreadPool’s 1 item (thread) is used in the background
  - we are working on a higher abstraction level (compared to Thread directly)
- Possibilities / advantages
  - **Return type**
  - **Non-blocking build-up (~*building item on top of each other*)**
  - **Exception handling**
  - **Stopping (cancelling)**

Task

- Older solution: **ThreadPool**
    - Using the ThreadPool directly
    - We have pre-defined threads which can be used and re-used
    - The number of these threads depends on the current load and current usage of the OS

```
static void Main(string[] args)
{
   ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);
}

private static void ComputeBoundOp(object state)
{  /* Long running operation */ }
```

- Disadvantages:
    - The threads are not directly available
    - No support for return types

Disadvantages can be solved by workarounds, but not the best way.

- **The Task instances are usually assigned to ThreadPool threads by the TaskScheduler**
    - If there are long operations and processes (long running task) then separate threads are created just for these
    - There are pre-defined APIs to handle the disadvantages

Task vs regular thread

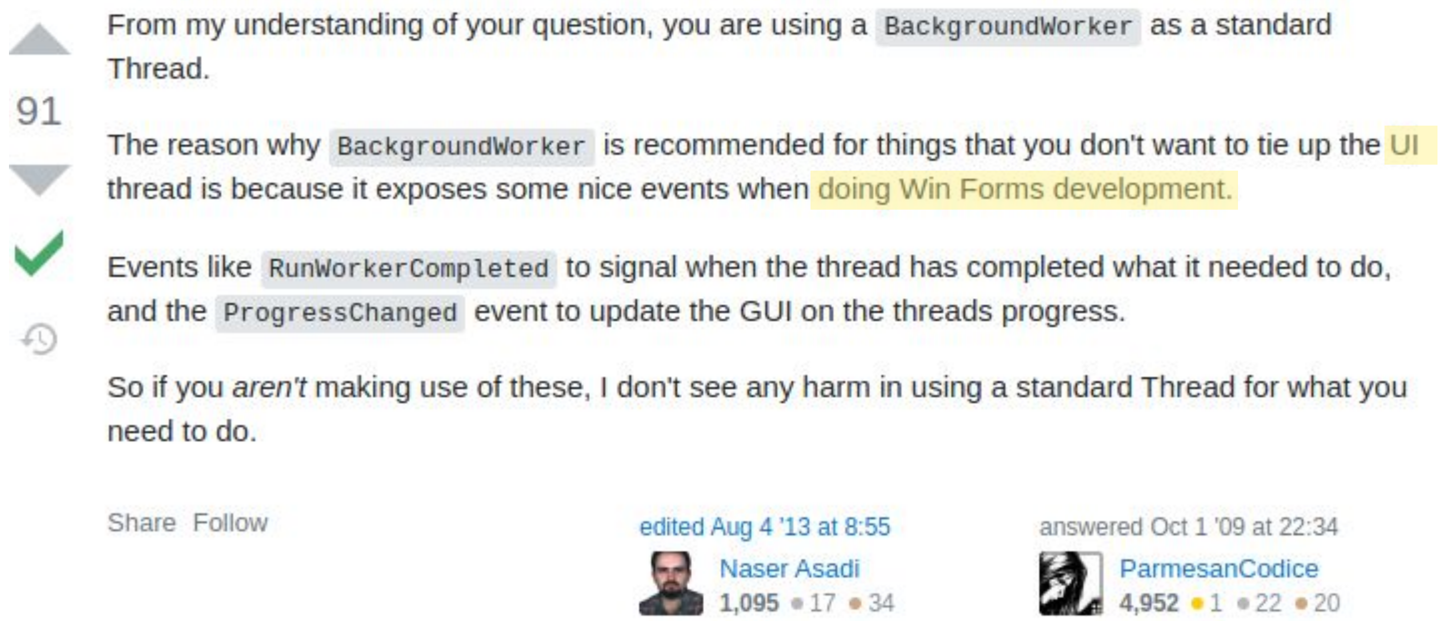
- Task advantages (“impossible” thread operations)
  - **Parameter handling**
    - Thread: nothing or object, always typecast
    - Task: custom type(s), compiler supported
  - **Returned-value handling**
    - Thread: nothing, difficult to return value to the main thread
    - Task: Task vs Task<T>
  - **Detection of end-of-operation, continue with other operation**
    - Thread: usually with events, hard to synchronize
    - Task: continuation, continue with a new Task
  - **Exception handling**
    - Thread: thread dies, not trivial to handle in other thread
    - Task: can be handled when reading the result or with continuation
  - (Async-Await: language-integrated parallelism – only with Task)

Task vs regular thread

- Use regular threads if:
  - When the thread must run with non-default priority
    - every ThreadPool thread has normal priority → usually not a problem
  - **A foreground thread is necessary**
    - every ThreadPool thread is a background thread – THE ONLY VALID REASON!
  - **The thread performs an extremely long operation**
    - e.g. Throughout the execution of the application – TaskCreationOptions.LongRunning
    - Thread vs BackGroundWorker [src]
  - **The usage of the immediate Abort() is required**
    - Try to avoid this in most cases
    - Abort vs Cancel
      - almost the same outcome, but when aborting the application can end up in an inconsistent state which is not good
      - using cancellation we can define exactly what should happen after the cancellation, so inconsistent states can be handled with ease [src]
- Almost in every (other) case: use Task
  - easy to use, robust tool
  - wide range of applications + additional possibilities (cancellation, exception handling, schedules etc.)

Thread vs BackGroundWorker

Thread vs BackGroundWorker [src]



91

From my understanding of your question, you are using a `BackgroundWorker` as a standard Thread.

The reason why `BackgroundWorker` is recommended for things that you don't want to tie up the UI thread is because it exposes some nice events when doing Win Forms development.

Events like `RunWorkerCompleted` to signal when the thread has completed what it needed to do, and the `ProgressChanged` event to update the GUI on the threads progress.

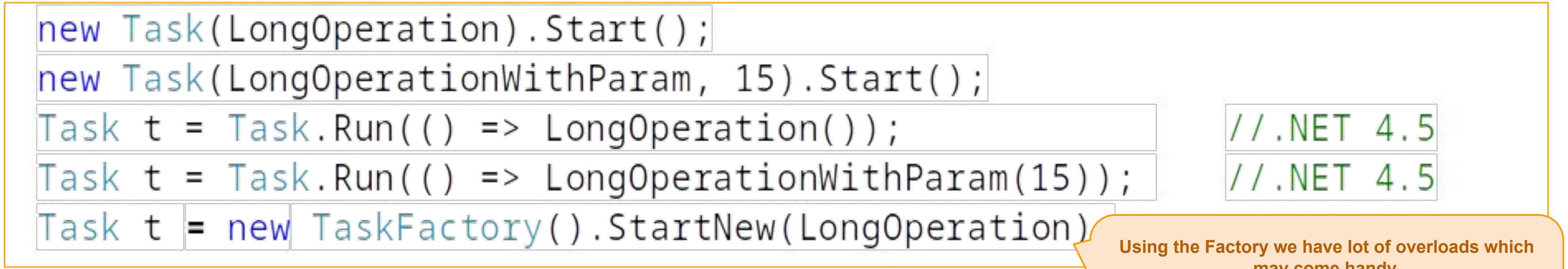
So if you *aren't* making use of these, I don't see any harm in using a standard Thread for what you need to do.

Share Follow

edited Aug 4 '13 at 8:55
Naser Asadi
1,095 ● 17 ● 34

answered Oct 1 '09 at 22:34
ParmesanCodice
4,952 ● 1 ● 22 ● 20

Task (launching)

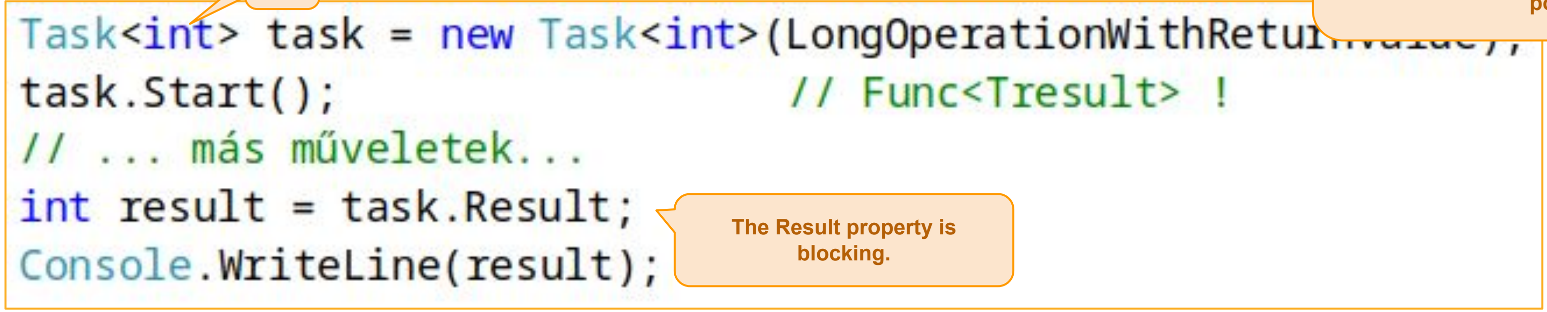


```
new Task(LongOperation).Start();
new Task(LongOperationWithParam, 15).Start();
Task t = Task.Run(() => LongOperation());                    //.NET 4.5
Task t = Task.Run(() => LongOperationWithParam(15));         //.NET 4.5
Task t = new TaskFactory().StartNew(LongOperation)
```

Using the Factory we have lot of overloads which may come handy.

(This current example does not show all possibilities.)

?!



```
Task<int> task = new Task<int>(LongOperationWithReturnValue);
task.Start();                   // Func<Tresult> !
// ... más műveletek...
int result = task.Result;
Console.WriteLine(result);
```

The Result property is blocking.

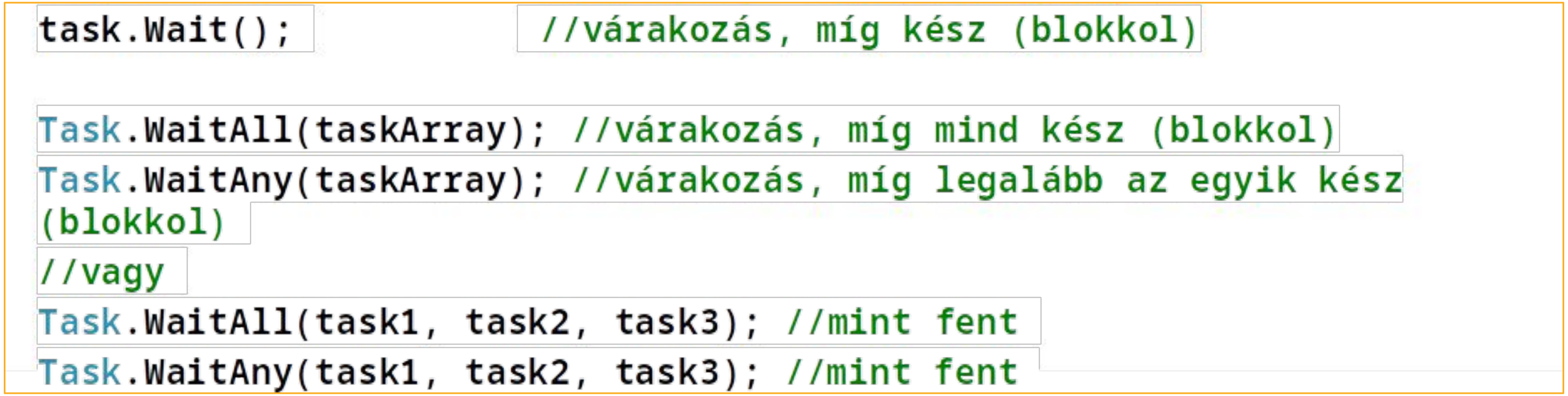
Task vs Thread return types

```
static void Main(string[] args)
{
    int result = 0;

    Thread t1 = new Thread(() => {
        Thread.Sleep(1500);
        result = 42;
    });
    t1.Start();
    Console.WriteLine("Feldolgozás elindult!");

    t1.Join();
    Console.WriteLine("Feldolgozás vége!");
    Console.WriteLine($"Eredmény: {result}");

    Console.ReadLine();
}
```

```
static void Main(string[] args)
{
    Task<int> t1 = new Task<int>(() => {
        Thread.Sleep(1500);
        return 42;
    });
    t1.Start();
    Console.WriteLine("Feldolgozás elindult!");

    t1.Wait();
    Console.WriteLine("Feldolgozás vége!");
    Console.WriteLine($"Eredmény: {t1.Result}");

    Console.ReadLine();
}
```

Result by alone is blocking, so there is no real need to call Wait beforehand.

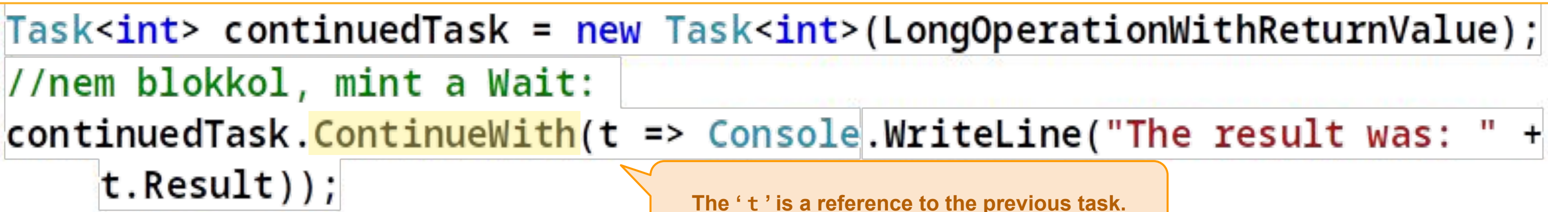
Task (synchronization)



```
task.Wait();             //várakozás, míg kész (blokkol)

Task.WaitAll(taskArray); //várakozás, míg mind kész (blokkol)
Task.WaitAny(taskArray); //várakozás, míg legalább az egyik kész
(blokkol)
//vagy
Task.WaitAll(task1, task2, task3); //mint fent
Task.WaitAny(task1, task2, task3); //mint fent
```

- In any case when we need to wait until the end of the task (~job).
- Eg. result calculation, array uploading, variable sets, file I/O operations

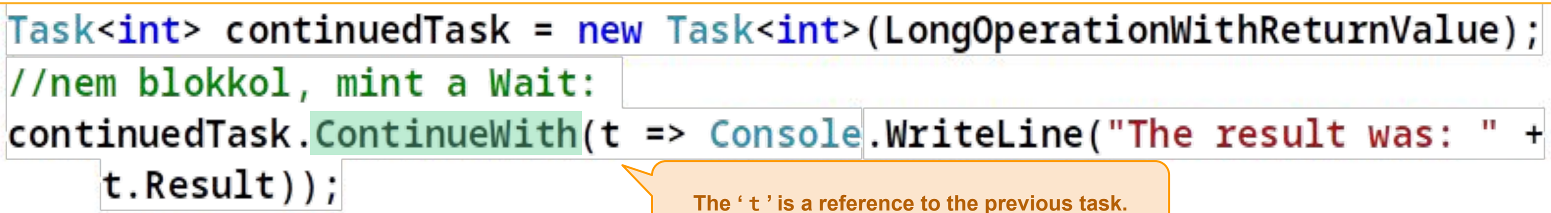
Task (continuation after finish)



```
Task<int> continuedTask = new Task<int>(LongOperationWithReturnValue);
//nem blokkol, mint a Wait:
continuedTask.ContinueWith(t => Console.WriteLine("The result was: " +
    t.Result));
```

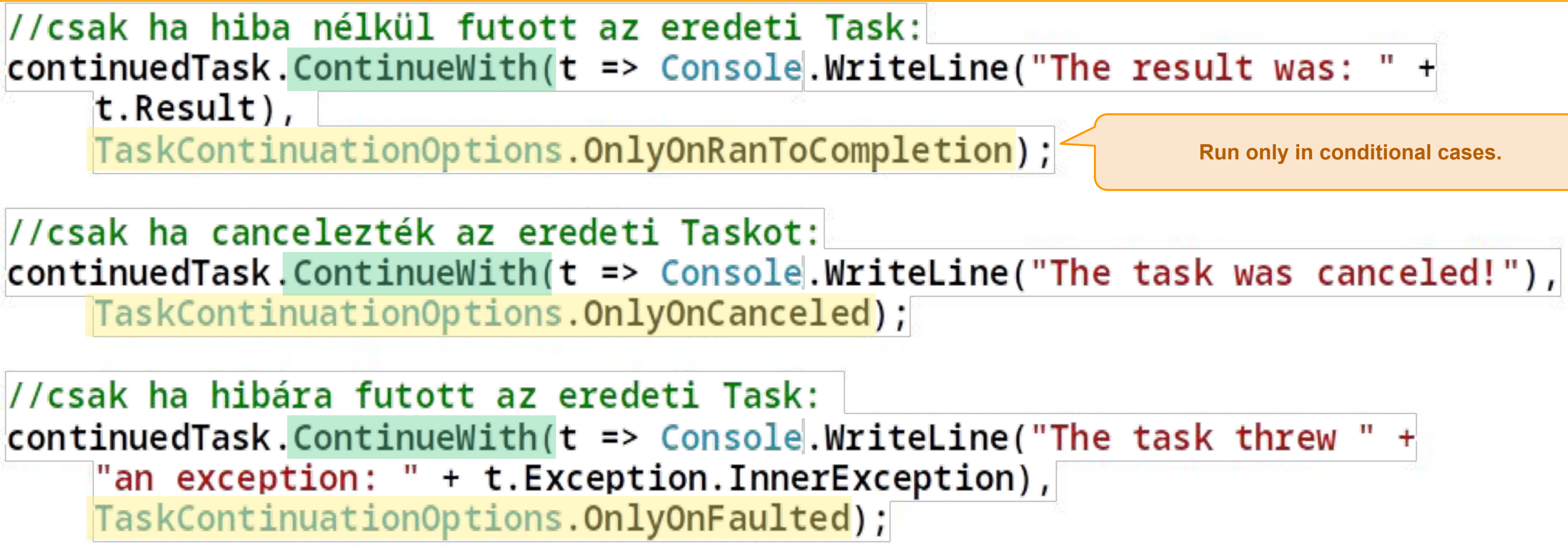
The ‘ t ’ is a reference to the previous task.

Task (continuation after finish)



```
Task<int> continuedTask = new Task<int>(LongOperationWithReturnValue);
//nem blokkol, mint a Wait:
continuedTask.ContinueWith(t => Console.WriteLine("The result was: " +
     t.Result));
```

The ‘ t ’ is a reference to the previous task.



```
//csak ha hiba nélkül futott az eredeti Task:
continuedTask.ContinueWith(t => Console.WriteLine("The result was: " +
     t.Result),
     TaskContinuationOptions.OnlyOnRanToCompletion);

//csak ha cancelezték az eredeti Taskot:
continuedTask.ContinueWith(t => Console.WriteLine("The task was canceled!"),
     TaskContinuationOptions.OnlyOnCanceled);

//csak ha hibára futott az eredeti Task:
continuedTask.ContinueWith(t => Console.WriteLine("The task threw " +
     "an exception: " + t.Exception.InnerException),
     TaskContinuationOptions.OnlyOnFaulted);
```

Run only in conditional cases.

Task (continuation after finish)

```
List<Task<int>> ts = new List<Task<int>>() {
    new Task<int>(() => { Thread.Sleep(800); return 800; }),
    new Task<int>(() => { Thread.Sleep(600); return 600; }),
    new Task<int>(() => { Thread.Sleep(1200); return 1200; }),
    new Task<int>(() => { Thread.Sleep(1100); return 1100; }),
};

Task.WhenAll(ts.ToArray()).ContinueWith(x => {
    //x.Result: int[]
    Console.WriteLine("Vége! Az eredmények összege: " +
x.Result.Sum());
});
```

`Task.WaitAll` blocks the current thread until everything has completed.

`Task.WhenAll` returns a task which represents the action of waiting until everything has completed.

Task (exceptions)

If there is an Exception in a Task, it is not raised.

When calling for Wait() or reading the Result, an **AggregateException** is thrown.

One Task can have multiple children Tasks, thus the multiple Exceptions.

```
Task t = new Task(() => {
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});

t.Start();
Console.WriteLine("Task elindult");
try { t.Wait(); }
catch (Exception e)
{
    Console.WriteLine("Baj történt, az alábbi üzenettel: " +
e.Message);
}
```

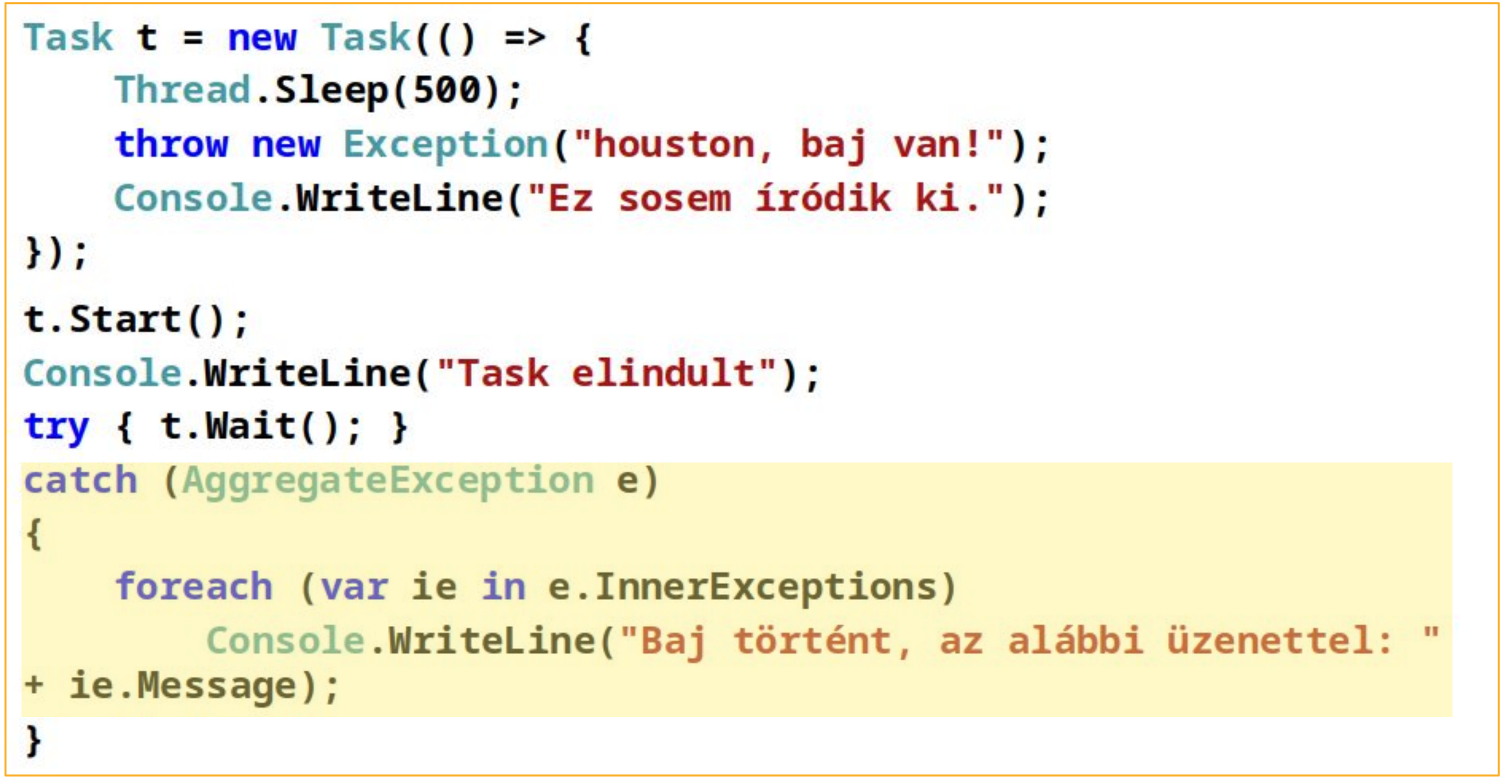
Task (exceptions)

If there is an Exception in a Task, it is not raised.

When calling for Wait() or reading the Result, an AggregateException is thrown.

One Task can have multiple children Tasks, thus the multiple Exceptions.

There is an InnerExceptions property, that is a collection for the actual Exception instances.



```
Task t = new Task(() => {
    Thread.Sleep(500);
    throw new Exception("houston, baj van!");
    Console.WriteLine("Ez sosem íródik ki.");
});
t.Start();
Console.WriteLine("Task elindult");
try { t.Wait(); }
catch (AggregateException e)
{
    foreach (var ie in e.InnerExceptions)
        Console.WriteLine("Baj történt, az alábbi üzenettel: "
+ ie.Message);
}
```

Task (exceptions)

Continuation only if there were problem (exception).

```
Task taskWithContinuation = new Task(LongOperationWithException);

taskWithContinuation.ContinueWith( t => {
    Console.WriteLine("Error happened: {0}", t.Exception.Message);
}, TaskContinuationOptions.OnlyOnFaulted);

taskWithContinuation.Start();
```

Task (canceling)

```
static void CancellableLongOperation(CancellationToken cancellationToken)
{
        for (int i = 0; i < 100 && !cancellationToken.IsCancellationRequested; i++)
        { Thread.Sleep(100); }
}
```

- The possibility of cancellation **must be handled inside the operation** of the Task
- No "Abort", so the code must be able to handle "cancel requests"
  - See Abort vs Cancel at previous slides
- The Task is notified via Synchronized **`CancellationToken`**, thus solved the common / shared variable problem
- How to check for cancellation:
  - **`cancellationToken.IsCancellationRequested`**
    - bool property
    - if true, we have to get out from the method
  - **`cancellationToken.ThrowIfCancellationRequested()`**
    - exception is raised if cancellation was requested, thus it exits the method
    - **advantage**: the exception obviously shows that there were some problem which resulted the stopping of the Task (instead of finishing the job)

Task (canceling)

```
static void CancellableLongOperation(CancellationToken cancellationToken)
{
        for (int i = 0; i < 100 && !cancellationToken.IsCancellationRequested; i++)
        { Thread.Sleep(100); }
}
```

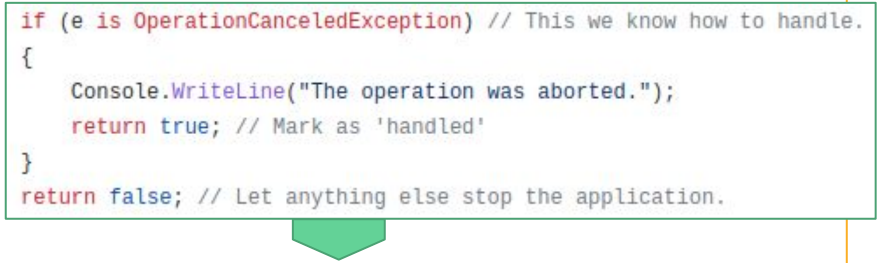
- `ae.Handle` marks the handled `InnerException` if the condition is true
- if there is any other exception then a new exception will be thrown
- if there is nothing else, then the C.W. line is executed
- using `CancellationToken.None` the cancellation can be disabled

```
CancellationTokenSource source = new CancellationTokenSource();
Task taskWithCancel = Task.Run(() => CancellableLongOperation(source.Token), source.Token);
source.Cancel();
try
{
        taskWithCancel.Wait();
}
catch (AggregateException ae)
{
        ae.Handle(e => { if(e is OperationCanceledException) { /* …CW… */ } });
}
```

Task (canceling - example 1.)

```
static void CancellableLongOperation(CancellationToken cancellationToken)
{
        for (int i = 0; i < 100 && !cancellationToken.IsCancellationRequested; i++)
        { Thread.Sleep(100); }
}
```

```
CancellationTokenSource source = new CancellationTokenSource();
Task taskWithCancel = Task.Run(() => CancellableLongOperation(source.Token), source.Token);
source.Cancel();
try
{
        taskWithCancel.Wait();
}
catch (AggregateException ae)
{
        ae.Handle(e => { if(e is OperationCanceledException) { /* … */ } });
}
```



```
if (e is OperationCanceledException) // This we know how to handle.
{
    Console.WriteLine("The operation was aborted.");
    return true; // Mark as 'handled'
}
return false; // Let anything else stop the application.
```

Task (canceling - example 2.)

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        if (cts.Token.IsCancellationRequested)
            return;
    }
}, cts.Token);

t.Start();
Console.WriteLine("Task elindult!");
Thread.Sleep(1200);
Console.WriteLine("Leállítás kezdeményezése...");
cts.Cancel();
```

Task (canceling - example 3.)

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        cts.Token.ThrowIfCancellationRequested();
    }
}, cts.Token);

t.Start();
Console.WriteLine("Task elindult!");
Thread.Sleep(1200);
Console.WriteLine("Leállítás kezdeményezése...");
cts.Cancel();
```

```
try { t.Wait(); }
catch (AggregateException e) {
    e.Handle(ie => {
        if (ie is OperationCanceledException)
            Console.WriteLine("Leállítva!");
        return ie is OperationCanceledException;
    });
}
```

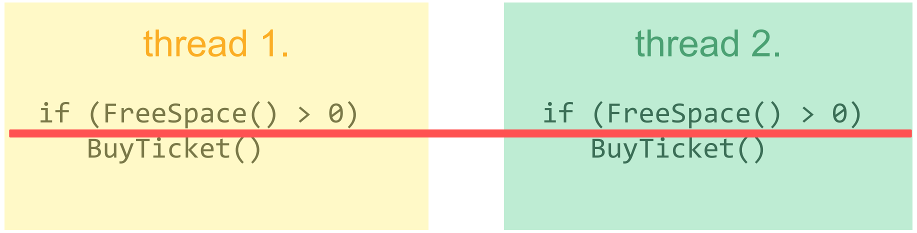
Task (canceling - example 4.)

```
CancellationTokenSource cts = new CancellationTokenSource();
Task t = new Task(() => {
    for (int i = 0; i < 1000; i++)
    {
        Thread.Sleep(10);
        Console.Write(i + "\t");
        cts.Token.ThrowIfCancellationRequested();
    }
}, cts.Token);

t.ContinueWith(x => { Console.WriteLine("Leállítva!"); },
TaskContinuationOptions.OnlyOnCanceled);

t.Start();
Console.WriteLine("Task elindult!");
Thread.Sleep(600);
Console.WriteLine("Leállítás kezdeményezése...");
cts.Cancel();
```

Multi-threaded phenomenons

Race condition (the problem)

- **Threads are competing with each other for resources.**
- Example:
  - You'd like to watch a movie in a cinema.
  - At the pay desk you ask if there is any available space.
  - You received yes as answer.
  - You go to a bit away, return back to buy tickets and you notified that there is no free space left.
  - ➔ **Between the checking (1st asking) and the action (buying the ticket) the number of free spaces changed, due to a different thread (other persons at different pay desk bought the tickets.)**

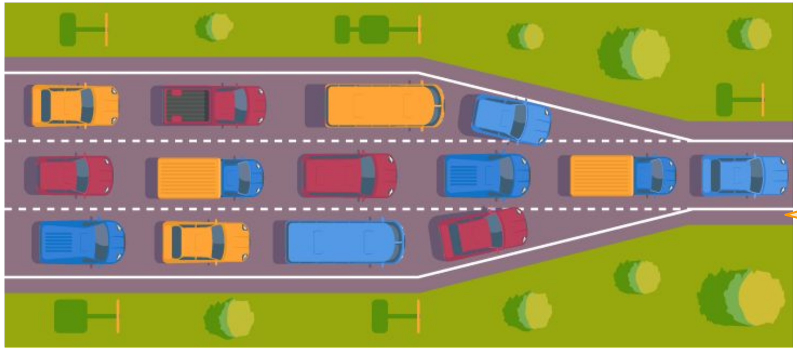


```
if (FreeSpace() > 0)
    BuyTicket()
```

```
if (FreeSpace() > 0)
    BuyTicket()
```

Race condition (the problem)

- Order of instruction execution
  - If threads are not using shared variables (/resources)
    - No problem, no logically hard step
    - If threads are using the same but on different data ➔ **data parallelism**
  - If threads are using shared variables (/resources)
    - Order of execution of the instructions MATTER
    - **Interleaving** can mean problem



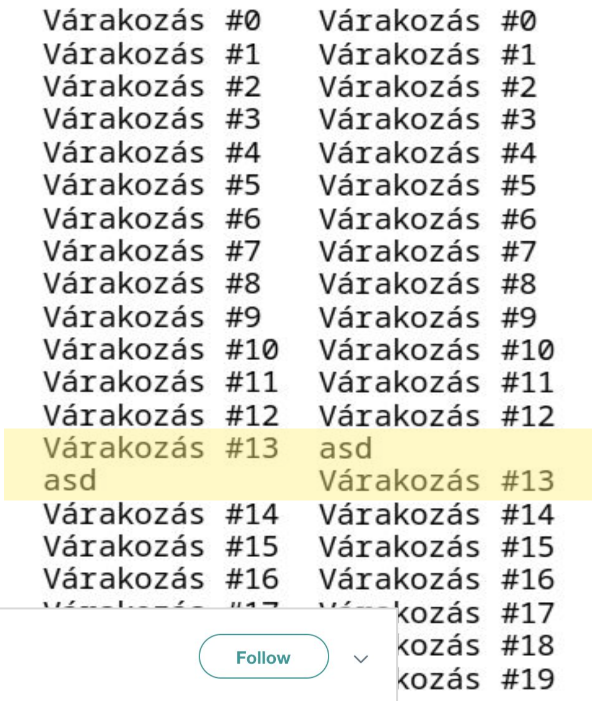
Race condition: who can run firstly from the waiting ones?

Race condition (the problem)

Example:

- write out something to the console using multiple threads
- console is 1 threaded environment ➜ the launched threads will compete for this, as the resource
- **race condition** is “happened”, which thread can use the console?
- **we can assume the execution order, but can not guarantee it** ⇒
  - Order in programs written using parallel concept is not (always) important.

```
Thread t = new Thread(() => {
    Thread.Sleep(1500);
    Console.WriteLine("asd");
});
t.Start();

for (int i = 0; i < 20; i++)
{
    Thread.Sleep(100);
    Console.WriteLine("Várakozás #{0}", i);
}

Console.ReadLine();
```



```
Várakozás #0    Várakozás #0
Várakozás #1    Várakozás #1
Várakozás #2    Várakozás #2
Várakozás #3    Várakozás #3
Várakozás #4    Várakozás #4
Várakozás #5    Várakozás #5
Várakozás #6    Várakozás #6
Várakozás #7    Várakozás #7
Várakozás #8    Várakozás #8
Várakozás #9    Várakozás #9
Várakozás #10   Várakozás #10
Várakozás #11   Várakozás #11
Várakozás #12   Várakozás #12
Várakozás #13   asd
asd             Várakozás #13
Várakozás #14   Várakozás #14
Várakozás #15   Várakozás #15
Várakozás #16   Várakozás #16
                  kozás #17
                  kozás #18
                  kozás #19
```

I Am Devloper
@iamdevloper

Follow

Knock knock
Race condition
Who's there?

12:07 PM - 11 Nov 2013

2,504 Retweets **1,013** Likes

38 2.5K 1.0K

Race condition (the problem)

```
static int sum = 0;

static void Main(string[] args)
{
    Thread th01 = new Thread(Szamol);
    Thread th02 = new Thread(Szamol);
    th01.Start();
    th02.Start();
    th01.Join();
    th02.Join();
    Console.WriteLine("Sum:  " + sum);
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        sum = sum + 1;
    }
}
```

Shared resource.

```
Sum:  54942
```

```
Sum:  56444
```

```
Sum:  82556
```

```
Sum:  100000
```

```
Sum:  66394
```

This is the anticipated result!
2 threads are counting up to
50.000 = 100.000.

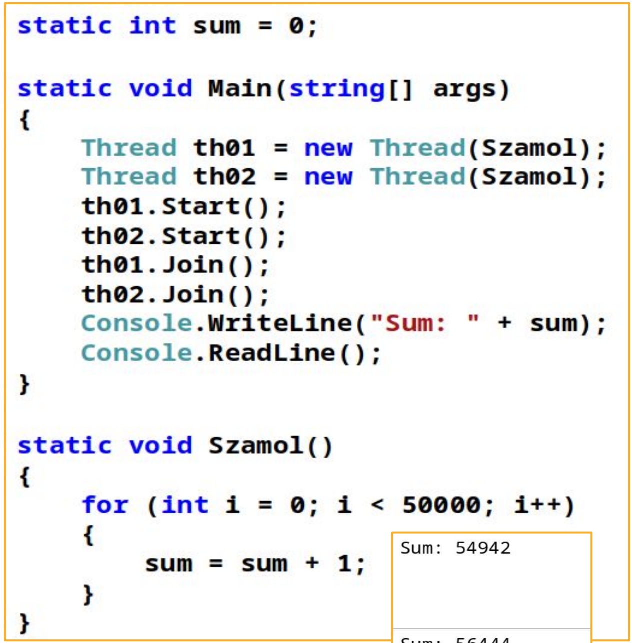
```
static int sum = 0;

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
            x => Console.WriteLine("Sum:  " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        sum = sum + 1;
    }
}
```

```
Sum:  56444
```

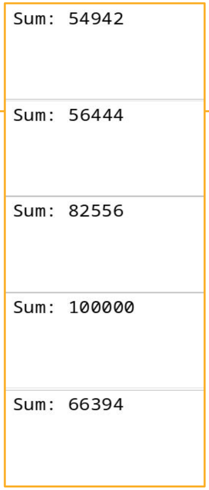
```
Sum:  82556
```

Race condition (the problem)

- What happens in the background?
  - The threads are working on the shared resources in a special way, that each of them “overwrites” the variables.
  - This results that the outcome is chaotic.
  - It is possible because in the background a simple variable assignment or incrementation builds up from multiple steps, like: x++ ⇒ read ➔ increment ➔ write back



```
static int sum = 0;

static void Main(string[] args)
{
    Thread th01 = new Thread(Szamol);
    Thread th02 = new Thread(Szamol);
    th01.Start();
    th02.Start();
    th01.Join();
    th02.Join();
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        sum = sum + 1;
    }
}
```



```
Sum: 54942
```

```
Sum: 56444
```

```
Sum: 82556
```

```
Sum: 100000
```

```
Sum: 66394
```

A = B = 0

thread 1.	thread 2.
r2 = A	r1 = B
B = 1	A = 2

r2 == 2 & r1 == 1

Deriving the modification of variables

	Thread 1	**Thread 2**

Original code
Eredeti kód
	t = x
	x = t + 1
		u = x
		x = u + 2

Possible outcome #1
Lehetséges kimenet #1
	(x is 0)
	t = x
	
	x = t + 1 **(x is 1)**
		
		u = x
		x = u + 2 (x is 2)

Possible outcome #2
Lehetséges kimenet #2
	
	t = x
	x = t + 1
		(x is 0)
		u = x
		
		x = u + 2 **(x is 2)**

Possible outcome #3
Lehetséges kimenet #3
	(x is 0)
	t = x
	x = t + 1 (x is 1)
		
		u = x
		x = u + 2 **(x is 3)**

Mutual exclusion (race condition solution I.)

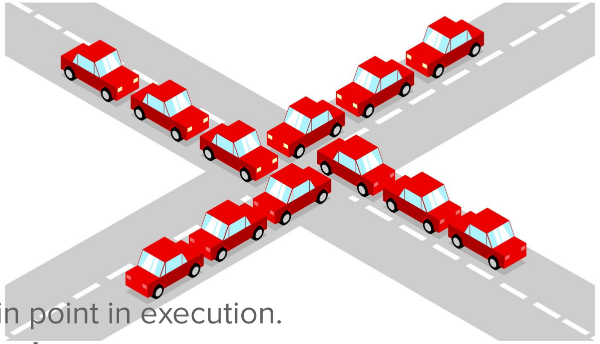
- Critical section:
  - The code segment which handles the **shared variables / resources** (eg. R/W).
  - **Any segment of the code where there are variables that cause a common dependency (shared) is called critical section.**
  - **There are multiple solutions to handle critical sections. All of them has the same aim: enable only 1 thread to enter (and thus be in) the critical section at any given time, exclusively.**
  - The main challenge to create or choose a solution, which safe and efficient in the same time.
- Mutual exclusion:
  - **Grants that the critical section can be entered only by 1 thread, exclusively, at the same time.**
  - provides a semantically correct programming structure to avoid competitive situations (race conditions)
- To achieve mutual exclusion in practice synchronization objects are used:
  - **lock**, **semaphore**, **critical section**, condition variable, event, **atomic & interlocked**
- How they work:
  - one thread “holds” the sync. object, while the other threads are must wait for the object to be free (released)
  - when the thread is done, it releases the sync. object
  - the other thread(s) can get the sync. object, lock it and do the same
- Example: bank safe ➔ the clerk grants that only 1 customer can be in the safe in the same time

Synchronization (race condition solution II.)

- **Synchronization can determine the execution order of the tasks; the determination of the relative order.**
- **Synchronization is an arbitrary mechanism that creates a reliable, predictable, deterministic relationship between the execution of two parallel tasks.**
- Often focused around commonly used data.
- In professional literature it is also called as randevou.
- Two main types
  - mutual exclusion „mutex”
  - conditional synchronization
- How they work:
  - Threads are launched at a given point of our program.
  - At some point there could be a need for the threads to wait for each other, because thread A needs something from thread B to finish or to move forward. → At this point the waiting threads are **not doing any useful work**.
  - Usage in practice is with special objects, mostly with condition variable or with event.
  - More detailed example, see Thread’s last example for text data processing.

Synchronization (outlook extra)

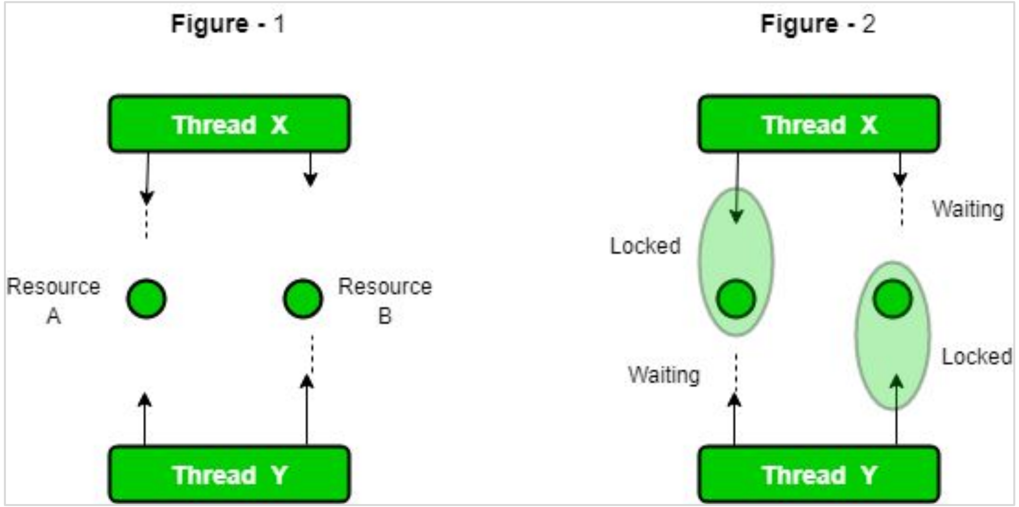
- The basic problem with parallel programming is the competition situation: between the execution of any two instructions, another thread may have access to the common data handled by (also) the previous thread.
  - For single-processor systems, the operating system scheduler (switching between threads) allows this, and for multiprocessor systems, the problem is even more common due to true (physical) concurrency. The different synchronization solutions are the main means of avoiding this (another related goal is to coordinate the timings).
- Synchronization is a mechanism that implements the interaction of parallel threads (or processes) that ensures the semantic correctness of the operations performed by the threads (or processes) in all circumstances.
  - Communicating threads running in parallel almost certainly requires the use of common resources (memory, ports, I / O devices, files). If their status is modified by a thread, but other threads have access to it, the latter threads can easily get erroneous or half-finished data.

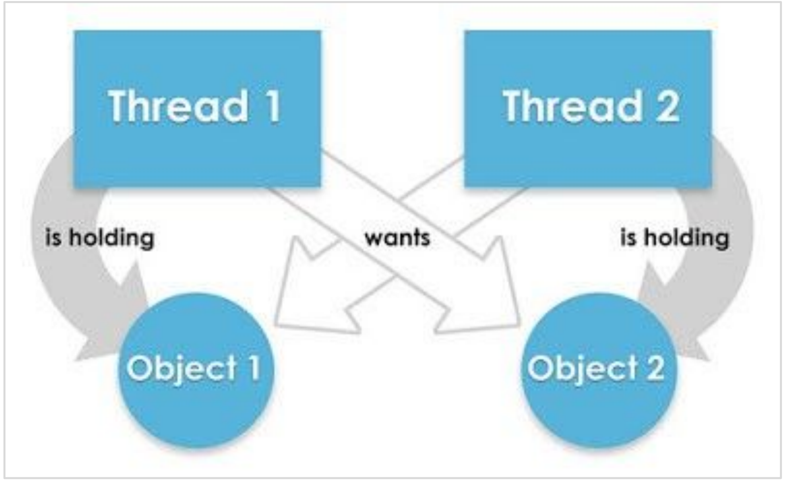


Deadlock (the problem)

- Concepts:
  - The processing subtasks of a parallel program cannot proceed at a certain point in execution.
  - **The threads are waiting for such an event (or condition) which will never happen.**
  - Deadlock is come to be when a thread is blocked and this block is never resolved.
  - So when a thread reaches a critical section but can never enter.
- Multiple deadlock can be defined based on the circumstances:
  - self-deadlock
    - the thread is waiting for such lock, which is already owned by itself
  - recursive deadlock
    - the thread which is in the critical section tries to reach a function which refers back to the locked section
  - lock-order deadlock
    - two threads, two locks ➔ see next slide
- Example:
  - traffic deadlock, see image
- Possible solution:
  - timeout usage
  - locking in right order

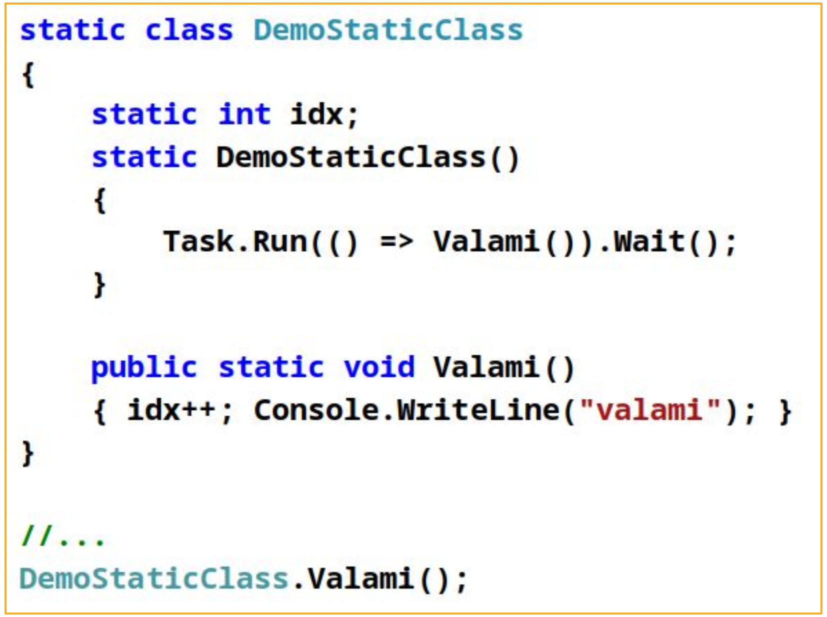
Deadlock (the problem)





Deadlock (example 1.)

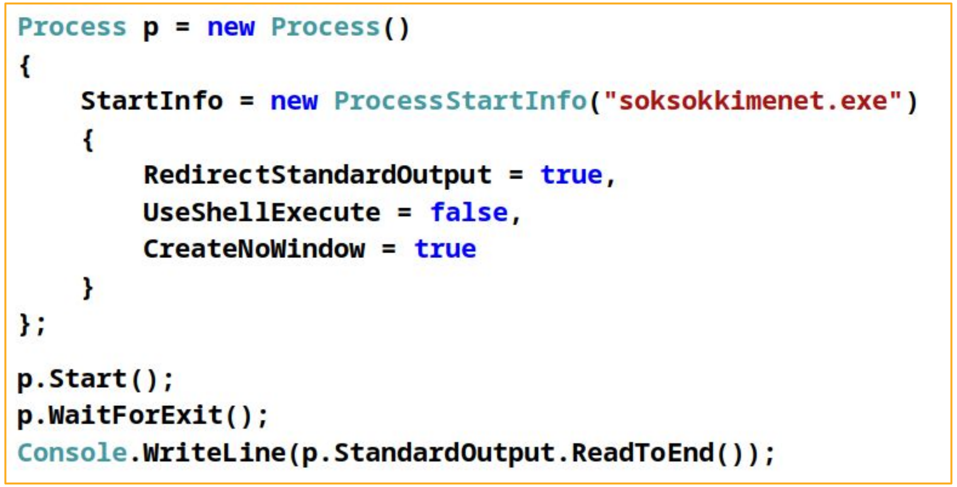
- The static ctor runs before the 1st access of the class.
- Before the DemoStaticClass.Valami() the static ctor should run.
- Which again refers to the Valami() method and waits for it, which blocks since the static ctor has not yet finished.
- The result is deadlock.



```
static class DemoStaticClass
{
    static int idx;
    static DemoStaticClass()
    {
        Task.Run(() => Valami()).Wait();
    }

    public static void Valami()
    { idx++; Console.WriteLine("valami"); }
}

//...
DemoStaticClass.Valami();
```

Deadlock (example 2.)

- The processes’ redirected output has a maximum buffer limit.
- If the buffer is full and the process tries to write on top of it, it has to wait until the buffer is not emptied.
- But in the meantime the host process waits for the (sub)process to finish and then only reads the buffer’s content.
- Thead result is deadlock.



```
Process p = new Process()
{
    StartInfo = new ProcessStartInfo("soksokkimenet.exe")
    {
        RedirectStandardOutput = true,
        UseShellExecute = false,
        CreateNoWindow = true
    }
};

p.Start();
p.WaitForExit();
Console.WriteLine(p.StandardOutput.ReadToEnd());
```

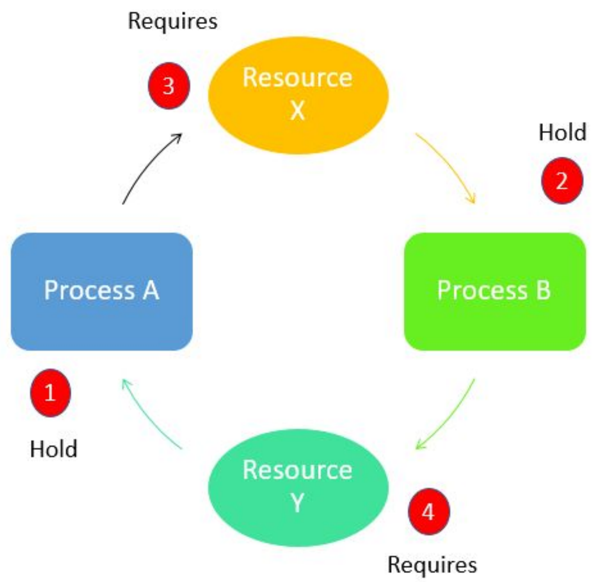
Solution for race condition and for deadlock

- A competitive situation exists when multiple threads try to access the same data at the same time
  - If possible, do not use shared objects for threads
    - additional caution for reference typed objects!
  - At certain tasks/data the value types can be considered
  - If it is not possible, use thread synchronization
  - But: shared data is acceptable IF the access is read-only for the threads

- To overcome deadlocks there is no real “best” solution, only thoughtful structuring and code writing.

Livelock (the problem)

- We speak of a live point if each subtask of the parallel program repeats the same (short) sequence of execution steps and is unable to pass them.
- A live point can occur when, due to nested resource reservations, processing does not stop, but is repeated indefinitely for all affected subtasks in a short period of time. It is much less common at a dead end, but even more difficult to treat.
- Examples of live points: in a chess game, the "perpetual chess".



Starvation

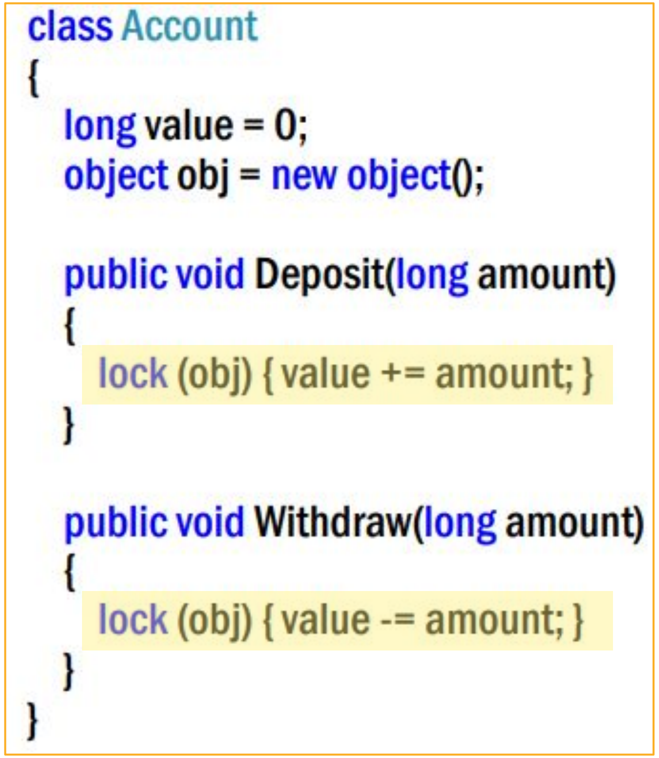
Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked. [src]

Starvation is a problem which is closely related to both, Livelock and Deadlock. In a dynamic system, requests for resources keep on happening. Thereby, some **policy is needed to make a decision about who gets the resource and when.** This process, being reasonable, may lead to a **some processes never getting serviced even though they are not deadlocked**. [src]

Lock object

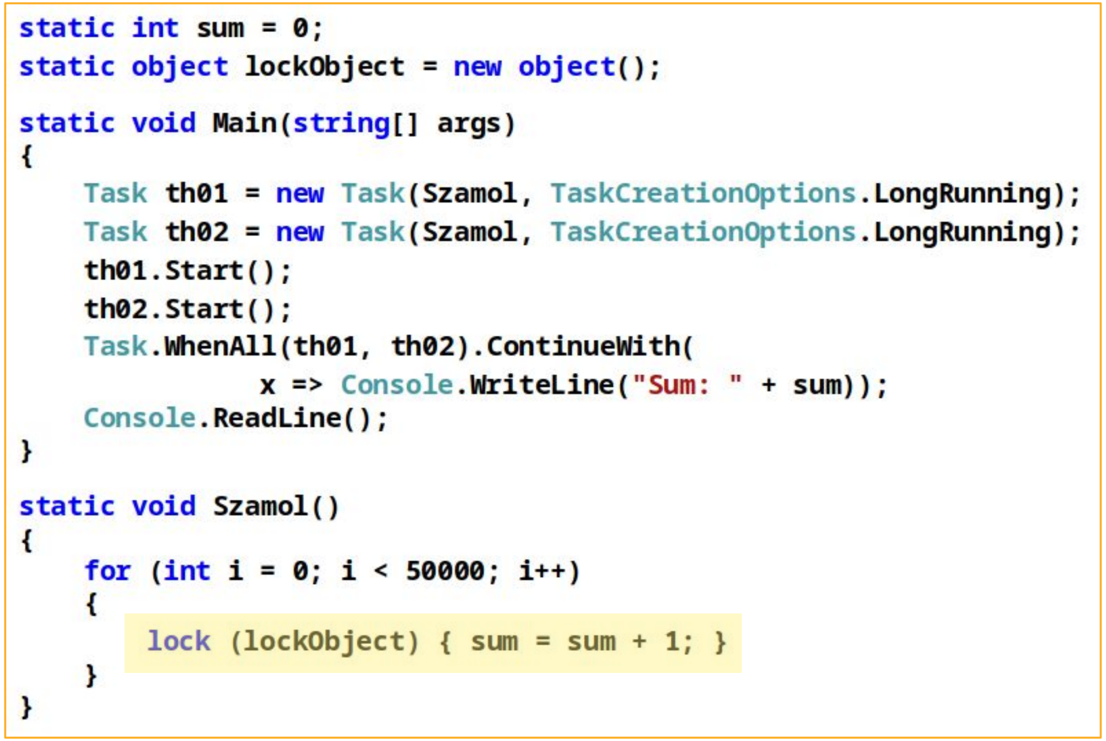
- lock (object o) { … } structure in C#
- implementation of mutual exclusion
- object o: synchronization object
    - can be any shared variable, but must be reference type
    - only 1 thread can own it concurrently
- the thread which owns the lock object can enter ({) to the critical section
- during exit (}) the lock object is released
- FIFO principle

Lock (example 1.)



```
class Account
{
   long value = 0;
   object obj = new object();

   public void Deposit(long amount)
   {
     lock (obj) { value += amount; }
   }

   public void Withdraw(long amount)
   {
     lock (obj) { value -= amount; }
   }
}
```

Lock (example 2.)



```
static int sum = 0;
static object lockObject = new object();

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
              x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
      lock (lockObject) { sum = sum + 1; }
    }
}
```

Lock (example 3.)

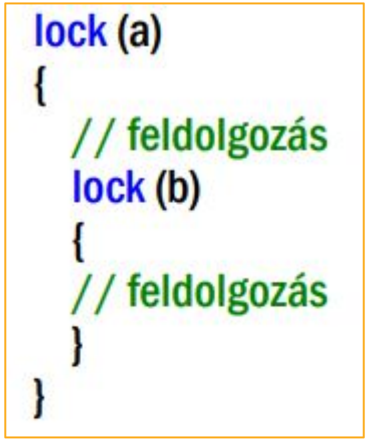
```
static int sum = 0;
static object lockObject = new object();

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
                x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    lock (lockObject)
    {
        for (int i = 0; i < 50000; i++)
        {
            sum = sum + 1;
        }
    }
}
```

Really important to only lock the least minimal part of the code!

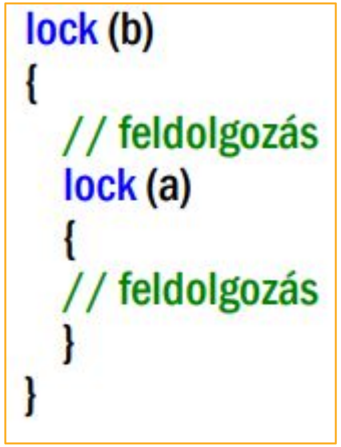
Create deadlock using locks

Classic deadlock example:

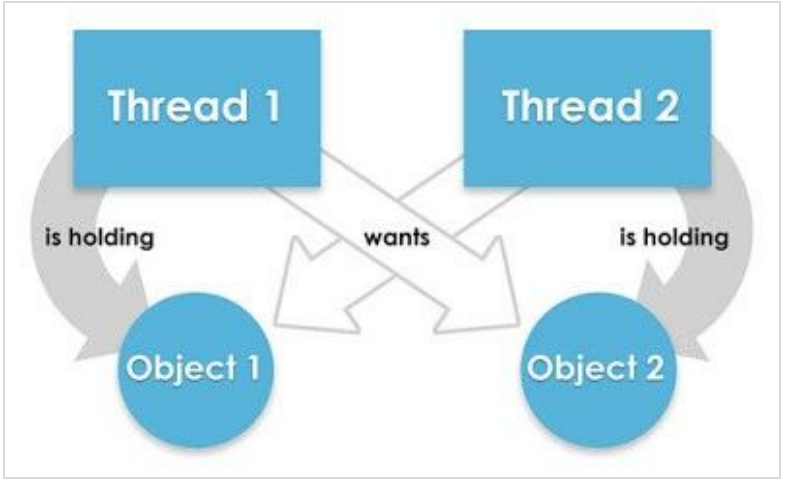
- We try to protect resources using lock. Both threads entered the 1st critical section, where they have to wait for the other thread's resource (which is locked by the other thread).



```
lock (a)
{
    // feldolgozás
    lock (b)
    {
    // feldolgozás
    }
}
```



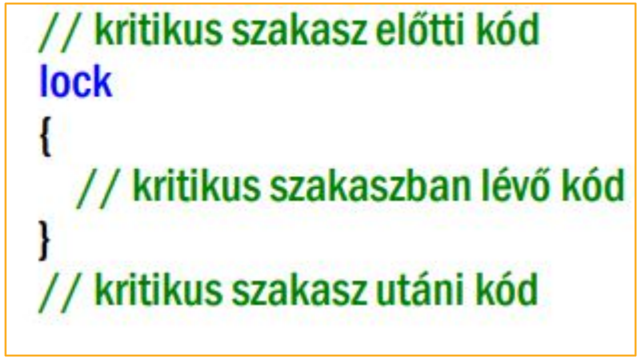
```
lock (b)
{
    // feldolgozás
    lock (a)
    {
    // feldolgozás
    }
}
```



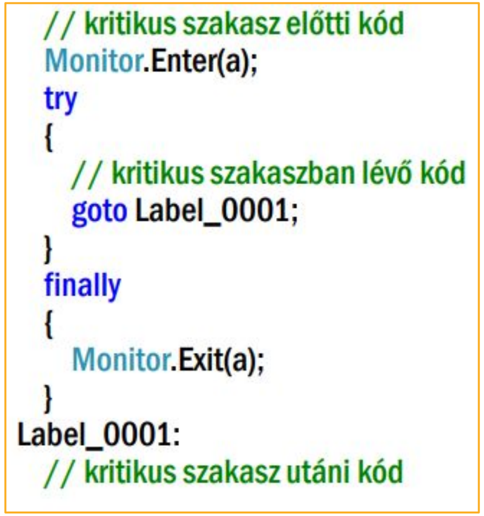
Monitor

The first task of the Monitor static class is to grant exclusive access to the given variables.

- Using this in the .NET framework both the mutual exclusion and the waiting or notification tied to the conditional variables, can be achieved.
- **The in-built "lock" is using the Monitor class in the background as well.**



```
// kritikus szakasz előtti kód
lock
{
    // kritikus szakaszban lévő kód
}
// kritikus szakasz utáni kód
```

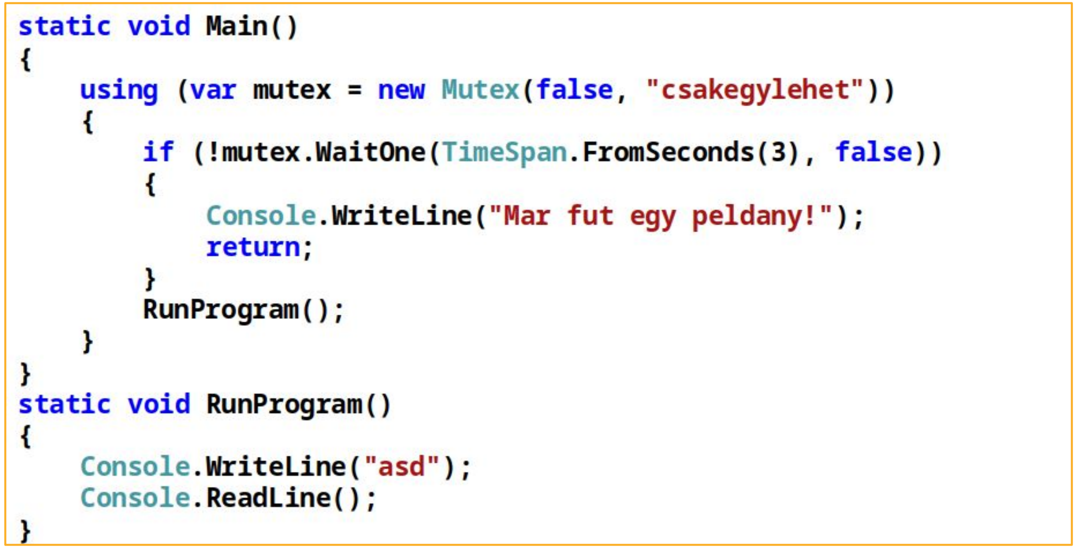


```
  // kritikus szakasz előtti kód
  Monitor.Enter(a);
  try
  {
     // kritikus szakaszban lévő kód
     goto Label_0001;
  }
  finally
  {
     Monitor.Exit(a);
  }
Label_0001:
  // kritikus szakasz utáni kód
```

Synchronization between processes

Mutex

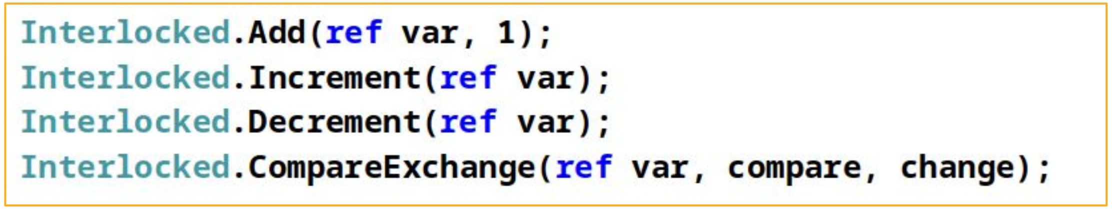
- Mutual exclusion between processes
- Mutex class



```
static void Main()
{
    using (var mutex = new Mutex(false, "csakegylehet"))
    {
        if (!mutex.WaitOne(TimeSpan.FromSeconds(3), false))
        {
            Console.WriteLine("Mar fut egy peldany!");
            return;
        }
        RunProgram();
    }
}
static void RunProgram()
{
    Console.WriteLine("asd");
    Console.ReadLine();
}
```

Atomicity

Sequence of operations can be considered as **atomic** which can not be further divided or interrupted. (See x++ ⇒ read ➔ increment ➔ write back)

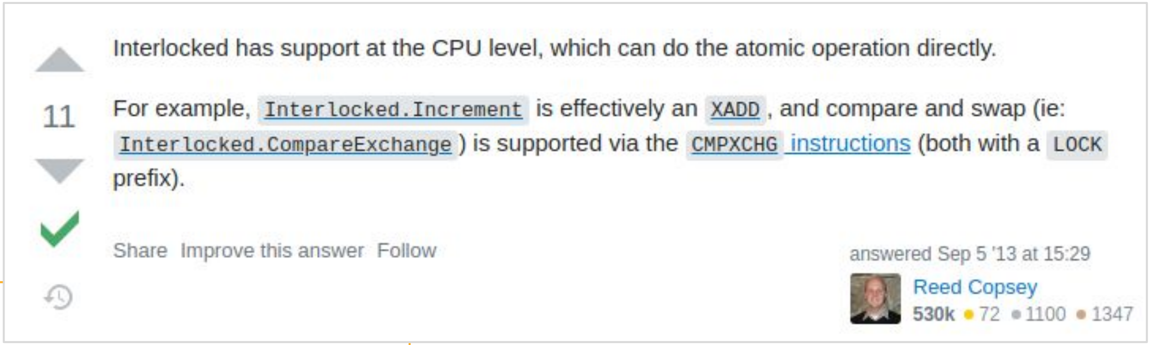


```
Interlocked.Add(ref var, 1);
Interlocked.Increment(ref var);
Interlocked.Decrement(ref var);
Interlocked.CompareExchange(ref var, compare, change);
```

Atomicitás (példa)

```
static int sum = 0;

static void Main(string[] args)
{
    Task th01 = new Task(Szamol, TaskCreationOptions.LongRunning);
    Task th02 = new Task(Szamol, TaskCreationOptions.LongRunning);
    th01.Start();
    th02.Start();
    Task.WhenAll(th01, th02).ContinueWith(
              x => Console.WriteLine("Sum: " + sum));
    Console.ReadLine();
}

static void Szamol()
{
    for (int i = 0; i < 50000; i++)
    {
        Interlocked.Increment(ref sum);
    }
}
```

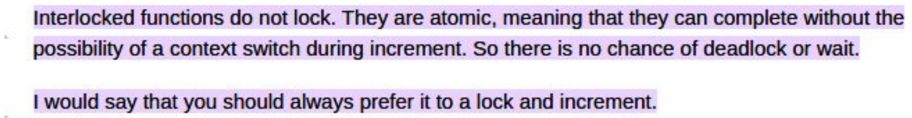
ref? Lock in the background?



Interlocked has support at the CPU level, which can do the atomic operation directly.

For example, `Interlocked.Increment` is effectively an `XADD`, and compare and swap (ie: `Interlocked.CompareExchange`) is supported via the `CMPXCHG` instructions (both with a `LOCK` prefix).

[src]

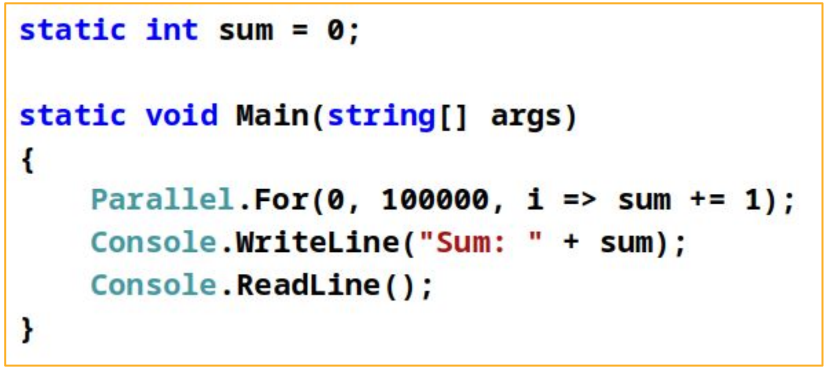


Interlocked functions do not lock. They are atomic, meaning that they can complete without the possibility of a context switch during increment. So there is no chance of deadlock or wait.

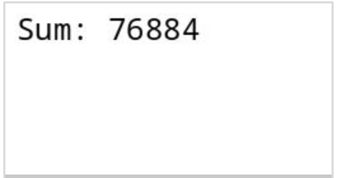
I would say that you should always prefer it to a lock and increment.

[src]

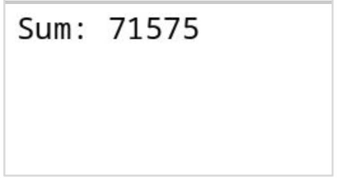
Parallel.For() and the race condition - example



```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => sum += 1);
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```

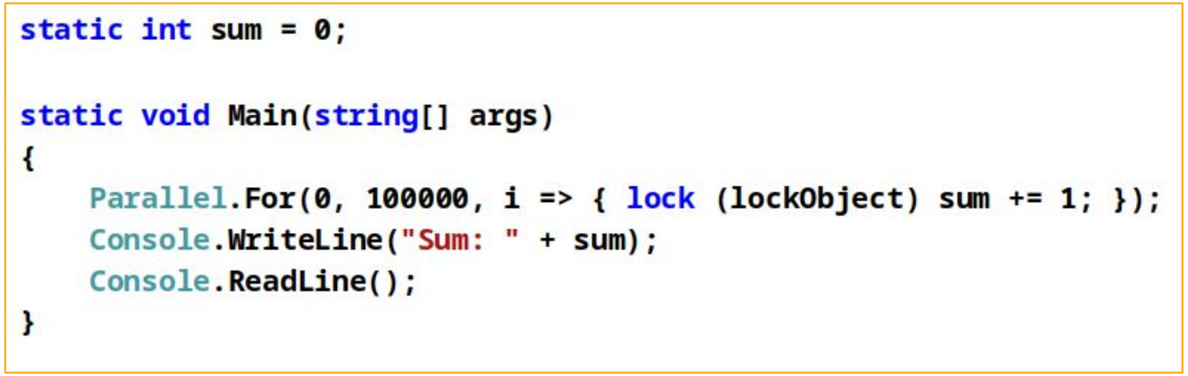


```
Sum: 76884
```

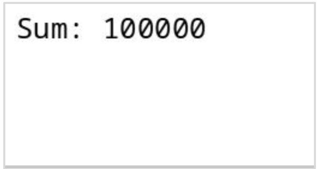


```
Sum: 71575
```

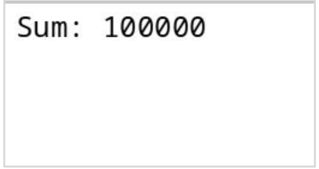
Parallel.For() and the lock - example



```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => { lock (lockObject) sum += 1; });
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```

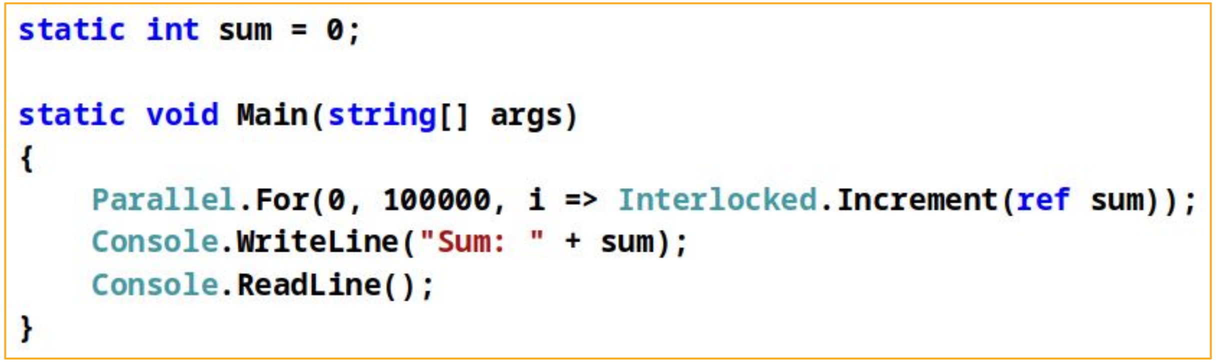


```
Sum: 100000
```

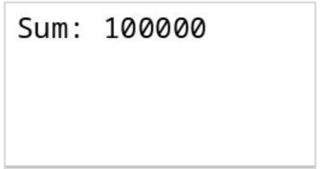


```
Sum: 100000
```

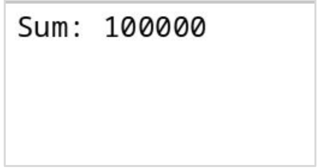
Parallel.For() and the atomic operation - example



```
static int sum = 0;

static void Main(string[] args)
{
    Parallel.For(0, 100000, i => Interlocked.Increment(ref sum));
    Console.WriteLine("Sum: " + sum);
    Console.ReadLine();
}
```



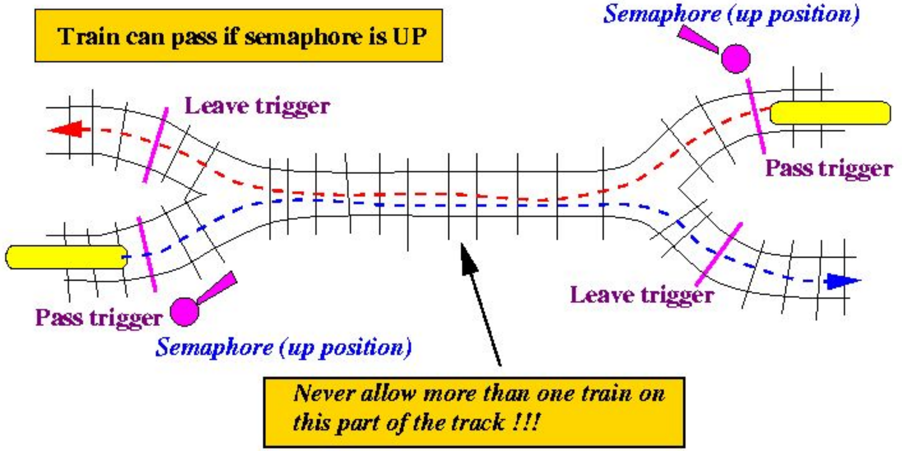
```
Sum: 100000
```



```
Sum: 100000
```

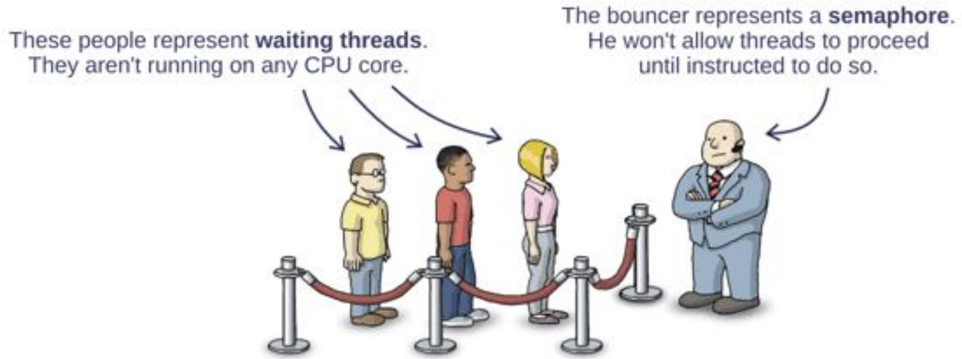
Semaphore

- Edsger Dijkstra, 1964
- Based on railway terminology
  - rail section is the critical section
  - the train can enter if the signal lever is open
  - if the trian enters the lever is set to closed
  - when the train exits the lever is set to open
  - ...repeat...
- Dijkstra described these with P() and V() atomic operations
- P(): during enter
- V(): during exit



Semaphore

- Atomic increment / decrement of an integer
  - During enter the value is decremented and checked
    - If the value is less than zero, then the thread is blocked
  - During exit the value is incremented
    - If there is waiting thread, then the next one is notified
- Can be used for **mutual exclusion** as well ➔ called binary semaphore
- But in the meantime can be used to determine **how many threads can use the critical section concurrently** ➔ an upper limit can be set


These people represent **waiting threads**.
They aren't running on any CPU core.
The bouncer represents a **semaphore**.
He won't allow threads to proceed
until instructed to do so.

Semaphore

```
static SemaphoreSlim _sem = new SemaphoreSlim(3);
static void Main()
{
    for (int i = 1; i <= 5; i++)
       new Thread(Enter).Start(i);
}
static void Enter(object id)
{
    Console.WriteLine(id + " wants to enter");
    _sem.Wait();
    Console.WriteLine(id + " is in!");
    Thread.Sleep(1000 * (int)id);
    Console.WriteLine(id + " is leaving");
    _sem.Release();
}
```

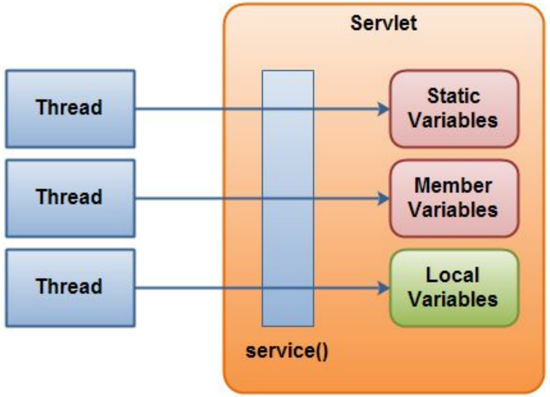
...specifying the initial number of requests that can be granted concurrently.

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Threadsafe

Thread safety: **A thread safe program protects its data from memory consistency errors.** In a highly multi-threaded program, a thread safe program does not cause any side effects with multiple read/write operations from multiple threads on same objects. **Different threads can share and modify object data without consistency errors.**

Resources that are thread-safe can be used by multiple threads without the risk of inconsistency. (Static resources are usually thread-proof, object instances are usually not.)



End of 2nd section.

Sources

Miklós Árpád, Dr. Vámossy Zoltán Design possibilities and methods of parallel algorithms lecture slides

Szabó-Resch Miklós Zsolt és Cseri Orsolya Eszter Advanced Programming lecture slides

Dr. Kertész Gábor Programming of Parallel and Distributed Systems lecture slides

MSDN

Sipos Miklós BPROF SzT specialization’s Advanced Development Techniques lecture slides

Sipos Miklós GitHub codes

Thanks for your attention!

Sipos Miklós

sipos.miklos@nik.uni-obuda.hu

https://users.nik.uni-obuda.hu/siposm/