План курса

0 – Параллелизм / Таксономия Флинна / Закон Амдала

1 – Многопоточность и асинхронность / Синхронизация / OpenMP

2 – Параллелизм на уровне процессов / MPI

3 – SIMD / GPGPU / CUDA

4 – MapReduce / Hadoop / mrjob / Hive

5 – Микросервисы / Очереди сообщений / RabbitMQ / Kafka

Фундаментальные понятия

Конкурентность (Concurrency)

Многопоточность (Multithreading)

Параллелизм (Parallelism)

Конкурентность

Concurrency is a very broad term, and it can be used at various levels:

Multiprocessing - Multiple Processors/CPUs executing concurrently. The unit of concurrency here is a CPU.

Multitasking - Multiple tasks/processes running concurrently on a single CPU. The operating system executes these tasks by switching between them very frequently. The unit of concurrency, in this case, is a Process.

Multithreading - Multiple parts of the same program running concurrently. In this case, we go a step further and divide the same program into multiple parts/threads and run those threads concurrently.

In simple terms, concurrency deals with managing the access to shared state from different threads and on the other side, parallelism deals with utilizing multiple CPUs or its cores to improve the performance of hardware.

Конкурентность

Low-Level Concurrency

In this level of concurrency, there is explicit use of atomic operations. Programmers don’ t code at this level, as it is very error-prone and difficult to debug.

Mid-Level Concurrency

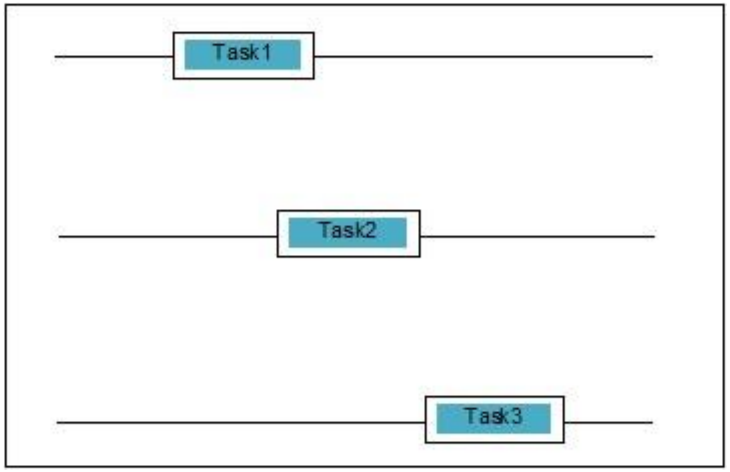
In this concurrency, there is no use of explicit atomic operations. It uses the explicit locks.

Examples: threads, sync primitives.

High-Level Concurrency

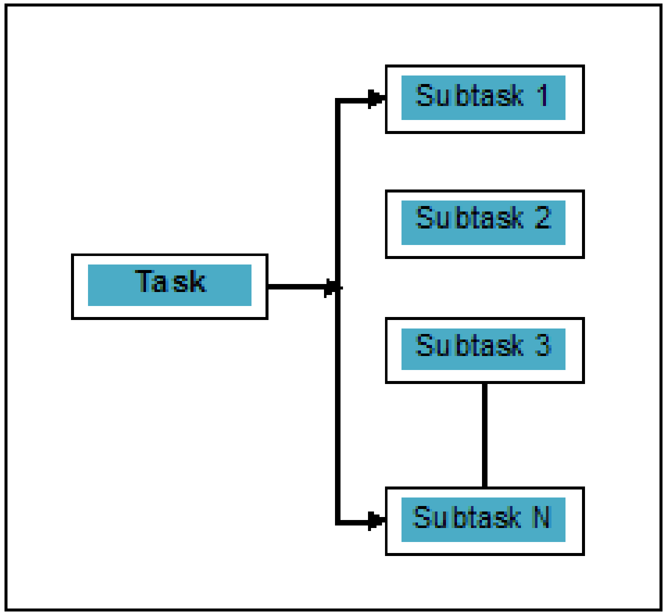
In this concurrency, neither explicit atomic operations nor explicit locks are used.

Examples: async/await.



Параллелизм

Parallelism may be defined as the art of splitting the tasks into subtasks that can be processed simultaneously. It is opposite to the concurrency, as discussed above, in which two or more events are happening at the same time.



Параллелизм

Concurrent but not parallel

An application can be concurrent but not parallel means that it processes more than one task at the same time but the tasks are not broken down into subtasks.

Parallel but not concurrent

An application can be parallel but not concurrent means that it only works on one task at a time and the tasks broken down into subtasks can be processed in parallel.

Neither parallel nor concurrent

An application can be neither parallel nor concurrent. This means that it works on only one task at a time and the task is never broken into subtasks.

Both parallel and concurrent

An application can be both parallel and concurrent means that it both works on multiple tasks at a time and the task is broken into subtasks for executing them in parallel.

Напомним: процесс

A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently.

A process typically contains the following resources:

• an image of the executable machine code associated with the program

memory, which includes:

- o executable code
- o process-specific data (input and output)
- o call stack that keeps track of active subroutines and/or other events
- o heap which holds intermediate computation data during run time

• operating system descriptors of resources that are allocated to the process such as file descriptors (unix/linux) and handles (windows), dat sources and sinks

• security attributes (process owner and set of permissions, e.g. allowable operations)

• processor state (context) such as registers and physical memory addressing

The operating system holds most of this information about active processes in data structures called process control blocks (PCB).

Напомним: поток

A thread is a basic unit of CPU utilization. It is also referred to as a "lightweight process".

A thread is a sequence of instructions within a process and it behaves like "a process within a process". It differs from a process because it does not have its own Process Control Block (collection of information about the processes). Usually, multiple threads are created within a process. Threads execute within a process and processes execute within the operating system kernel.

A thread comprises:

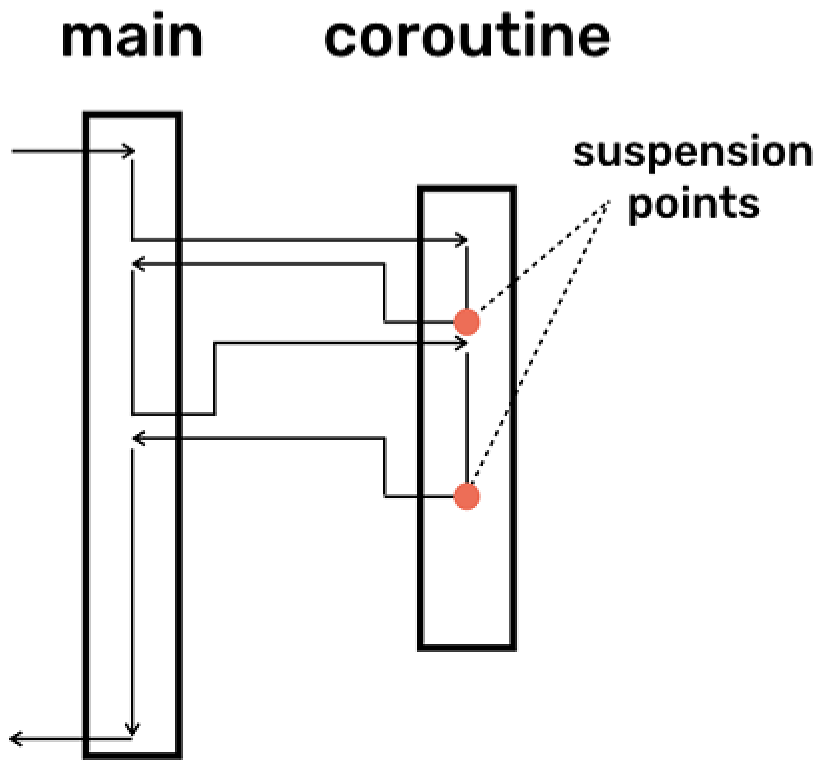
- thread ID
- program counter
- register set
- stack

A thread shares resources with its peer threads (all other threads in a particular task), such as:

- code section
- data section
- any operating resources which are available to the task

Напомним: корутина

It is a computer program that generalize subroutines to allow multiple entry points for suspending and resuming execution at certain locations. Coroutines are suitable for implementing cooperative tasks, iterators, infinite lists and pipes.



Объект Thread

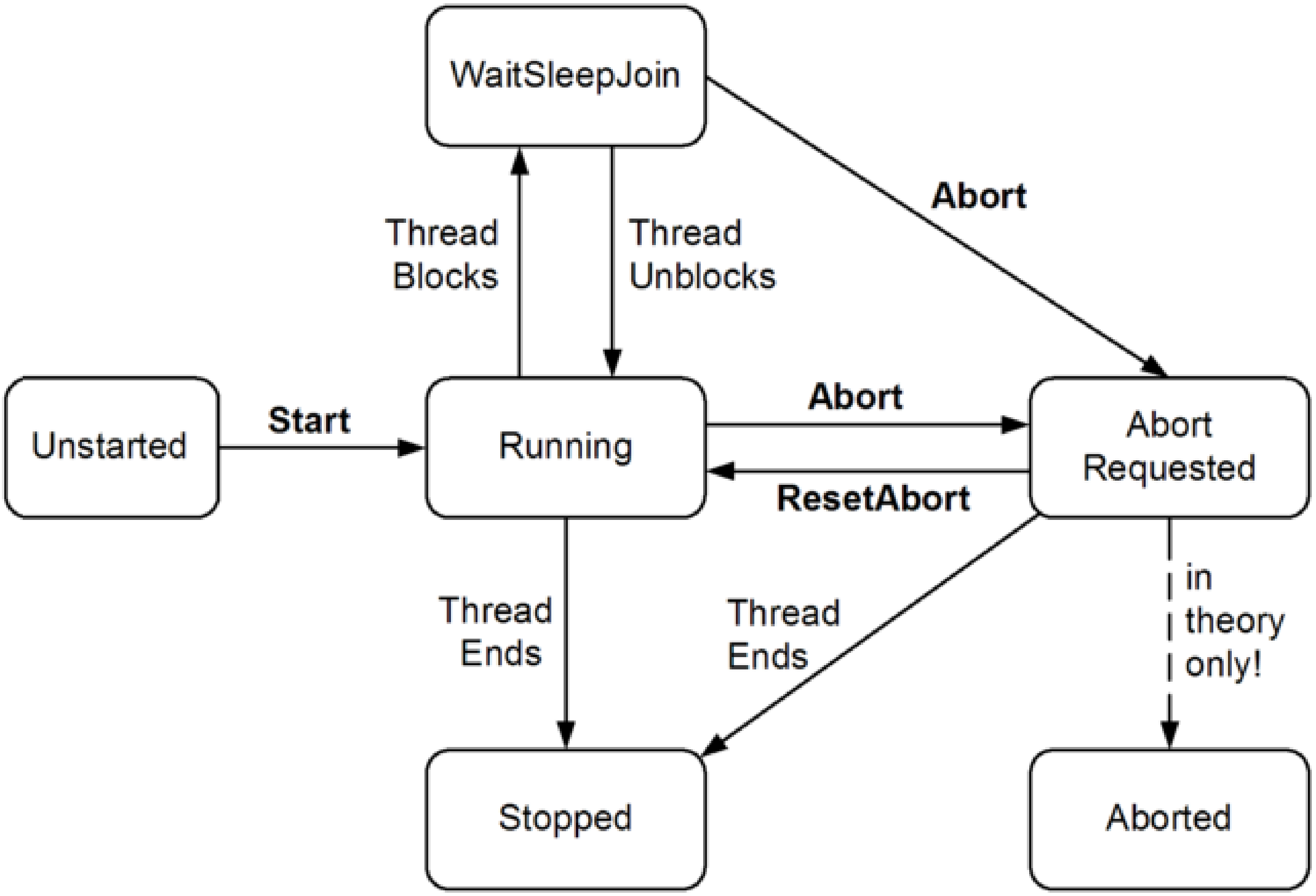
```
static void Main(string[] args)
{
    Thread myThread = new Thread(new ThreadStart(Count));
    myThread.Start();

    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Главный поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(300);
    }

    Console.ReadLine();
}

public static void Count()          // thread func
{
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("Второй поток:");
        Console.WriteLine(i * i);
        Thread.Sleep(400);
    }
}
```

Состояния потока

Состояния потока содержатся в перечислении **ThreadState**:

Aborted: поток остановлен, но пока еще окончательно не завершен
AbortRequested: для потока вызван метод Abort, но остановка потока еще не произошла
Background: поток выполняется в фоновом режиме
Running: поток запущен и работает (не приостановлен)
Stopped: поток завершен
StopRequested: поток получил запрос на остановку
Suspended: поток приостановлен
SuspendRequested: поток получил запрос на приостановку
Unstarted: поток еще не был запущен
WaitSleepJoin: поток заблокирован в результате действия методов Sleep или Join

Состояния потока



Приоритеты потока

Приоритеты потоков располагаются в перечислении ThreadPriority:

- Lowest
- BelowNormal
- Normal
- AboveNormal
- Highest

По умолчанию потоку задается значение Normal. Можно изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

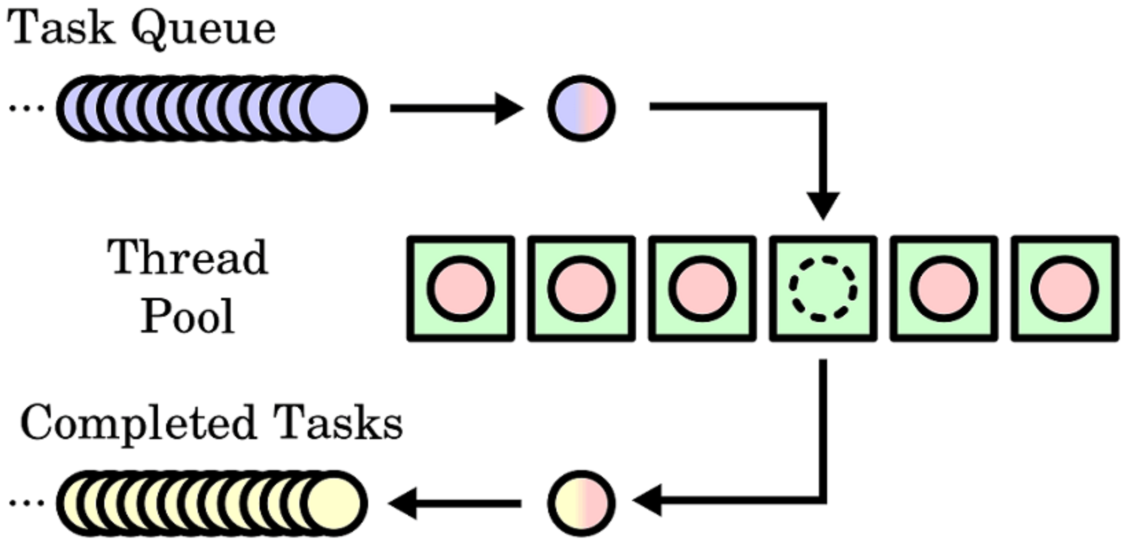
Объект Thread в C++

```
#include <thread>

void threadFunction()
{
    // do smth
}

int main()
{
    std::thread thread(threadFunction);
    thread.join();
    return 0;
}
```

Пул потоков

Если имеются небольшие задачи, которые требуют фоновой обработки, пул управляемых потоков - это самый простой способ воспользоваться преимуществами нескольких потоков.

Пул потоков – это паттерн, означающий, что при его создании разработчик получает определенное количество рабочих *background*-потоков, обрабатывающих запросы. При уничтожении самого пула потоки также удаляются. В результате экономятся ресурсы, которые до этого шли на создание и уничтожение потоков.



Пул потоков в CLR .NET

CLR thread pool is a group of <u>warmed up threads that are ready to be assigned work</u> to process.

The CLR Thread Pool contains 2 types of threads that have different roles.

1) Worker Threads
Worker threads are threads that process HTTP requests that come into your web server - basically they handle and process your application's logic.

2) Input/Output (I/O) Completion Port or IOCP Threads
These threads handle communication from your application's code to a network type resource, like a database or web service.

There is really no technical difference between worker threads and IOCP threads. The CLR Thread Pool keeps separate pools of each simply to avoid a situation where high demand on worker threads exhausts all the threads available to dispatch native I/O callbacks, potentially leading to a deadlock. However, this can still occur under certain circumstances.

TPL в .NET

Задача – это концептуальный объект, представляющий некоторую операцию, которая должна быть выполнена. Задача содержит в самой себе статус своей завершенности и сам результат, если она завершена.

В библиотеке классов .NET задача представлена специальным классом - классом Task, который находится в пространстве имен System.Threading.Tasks. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке.

```
// 1
Task task1 = new Task(() => Console.WriteLine("Task1 is executed"));
task1.Start();

// 2
Task task2 = Task.Factory.StartNew(() => Console.WriteLine("Task2 is executed"));

// 3
Task task3 = Task.Run(() => Console.WriteLine("Task3 is executed"));
```

TPL в .NET

```
Task task = new Task(Display);
task.Start();
task.Wait();
Console.WriteLine("Завершено");
```

Класс Task имеет ряд свойств, с помощью которых мы можем получить информацию об объекте. Некоторые из них:

AsyncState: возвращает объект состояния задачи
CurrentId: возвращает идентификатор текущей задачи
Exception: возвращает объект исключения, возникшего при выполнении задачи
Status: возвращает статус задачи

Task vs. Thread

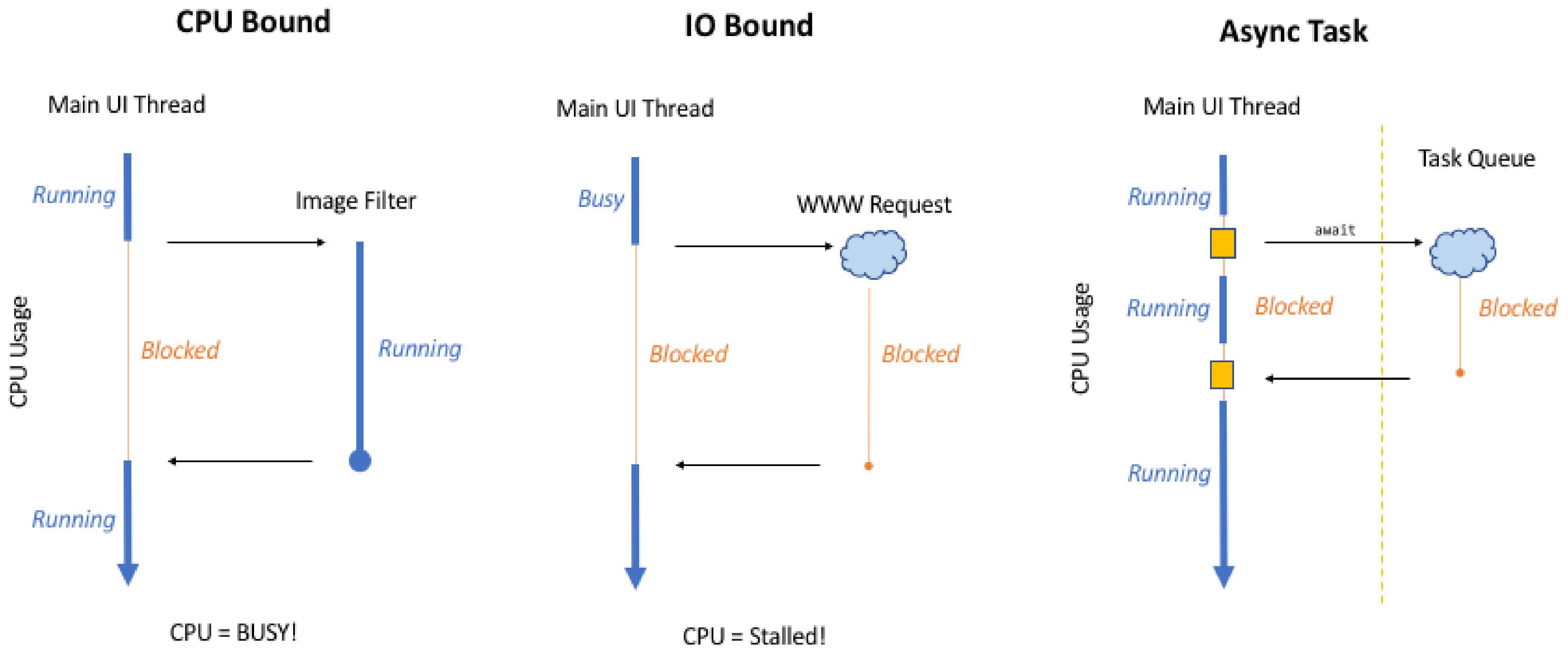
1. The Thread class is used for creating and manipulating a thread in Windows. A Task represents some asynchronous operation and is part of the TPL, a set of APIs for running tasks asynchronously and in parallel.
2. The task can return a result. There is no direct mechanism to return the result from a thread.
3. Task supports cancellation through the use of cancellation tokens. But Thread doesn't.
4. A task can have multiple processes happening at the same time. Threads can only have one task running at a time.
5. We can easily implement Asynchronous using ’ async’ and ‘await’ keywords.
6. A new Thread()is not dealing with Thread pool thread, whereas Task does use thread pool thread.
7. A Task is a higher level concept than Thread.

Task vs. Thread

```
Task.Factory.StartNew(ThreadFunction, TaskCreationOptions.LongRunning);
```

Данная строка кода максимально соответствует созданию выделенного потока с помощью “new Thread” , т.е. вне пула потоков, но все равно создается background-поток.

Асинхронность



Асинхронность

Футура (future) представляет собой обертку, над каким-либо значением или объектом (значением), вычисление или получение которого происходит отложено. Точнее, *future* предоставляет доступ к некоторому разделяемому состоянию, которое состоит из 2-х частей: данные (здесь лежит значение) и флаг готовности. *Футура* является получателем значения и не может самостоятельно выставлять его; роль *future* пассивна.

```
public async Task MethodWithReturnAsync()
{
    // ...
    var content = await GetUrlAsStringAsync("http://google.com");
    // ...
    // synchronous:
    // var content = GetUrlAsStringAsync("http://google.com").Result;
}
```

Асинхронность

Под *future* обычно имеется в виду представление переменной, доступное *только для чтения*, а под promise — изменяемый контейнер с одиночным присваиванием, который передаёт значение *future*. *Future* может быть определён без указания того, из какого *promise* будет получено значение. Также с одним *future* может быть связано несколько *promise*, однако присвоить значение future сможет только один promise.

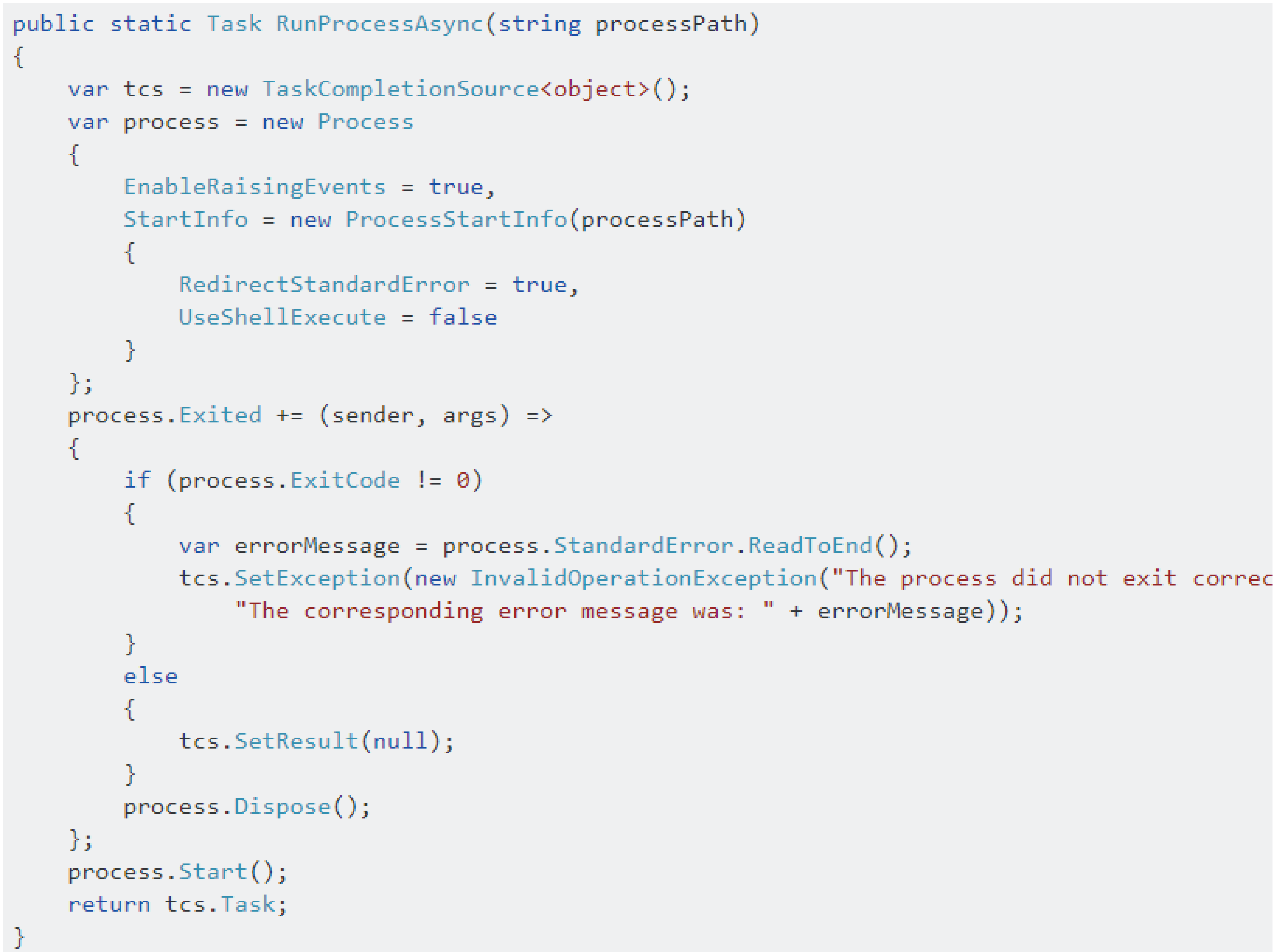
В остальных случаях *future* и *promise* создаются вместе и привязываются друг к другу: future — это значение, а promise — это функция, которая присваивает значение. На практике *future* — возвращаемое значение асинхронной функции *promise*. Процесс присваивания значения future называют resolving, fulfilling или binding.

Асинхронность в .NET

- `Task<T>` is a future (or `Task` for a unit-returning future).
- `TaskCompletionSource<T>` is a promise.

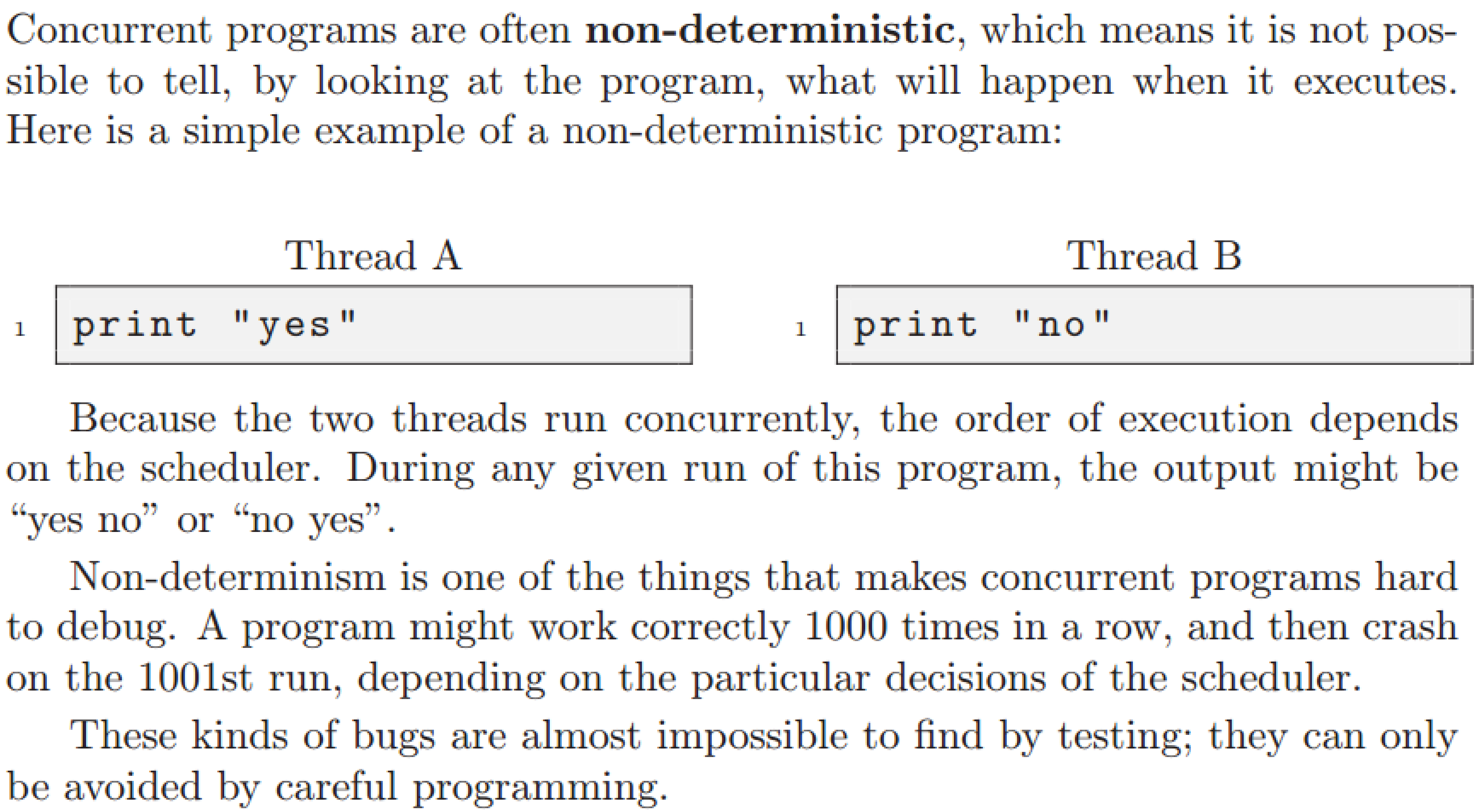
```
// var promise = new Promise<MyResult>;
var promise = new TaskCompletionSource<MyResult>();

// handlerMyEventsWithHandler(msg => promise.Complete(msg););
handlerMyEventsWithHandler(msg => promise.TrySetResult(msg));

// var myResult = promise.Future.Await(2000);
var completed = await Task.WhenAny(promise.Task, Task.Delay(2000));
if (completed == promise.Task)
  ; // Do something on timeout
var myResult = await completed;

Assert.Equals("my header", myResult.Header);
```

Асинхронность в .NET



```
public static Task RunProcessAsync(string processPath)
{
    var tcs = new TaskCompletionSource<object>();
    var process = new Process
    {
        EnableRaisingEvents = true,
        StartInfo = new ProcessStartInfo(processPath)
        {
            RedirectStandardError = true,
            UseShellExecute = false
        }
    };
    process.Exited += (sender, args) =>
    {
        if (process.ExitCode != 0)
        {
            var errorMessage = process.StandardError.ReadToEnd();
            tcs.SetException(new InvalidOperationException("The process did not exit correc
                "The corresponding error message was: " + errorMessage));
        }
        else
        {
            tcs.SetResult(null);
        }
        process.Dispose();
    };
    process.Start();
    return tcs.Task;
}
```

(Не)детерминизм



Concurrent programs are often **non-deterministic**, which means it is not possible to tell, by looking at the program, what will happen when it executes. Here is a simple example of a non-deterministic program:

Thread A

```
print "yes"
```

Thread B

```
print "no"
```

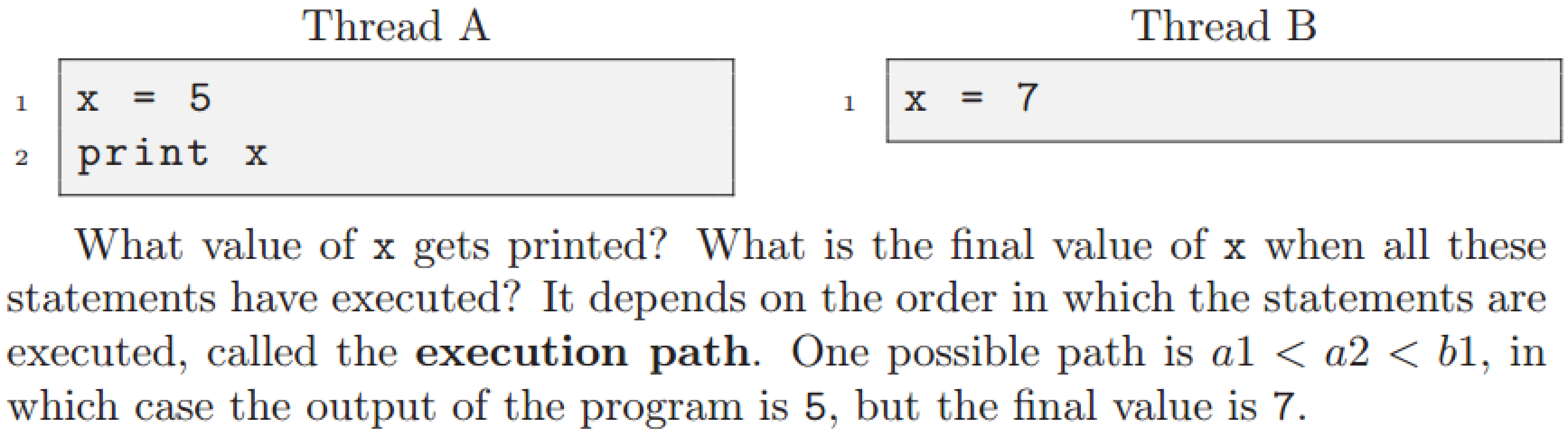
Because the two threads run concurrently, the order of execution depends on the scheduler. During any given run of this program, the output might be “yes no” or “no yes”.

Non-determinism is one of the things that makes concurrent programs hard to debug. A program might work correctly 1000 times in a row, and then crash on the 1001st run, depending on the particular decisions of the scheduler.

These kinds of bugs are almost impossible to find by testing; they can only be avoided by careful programming.

Разделяемая память

1) Конкурентная запись



Thread A

```
1  x = 5
2  print x
```

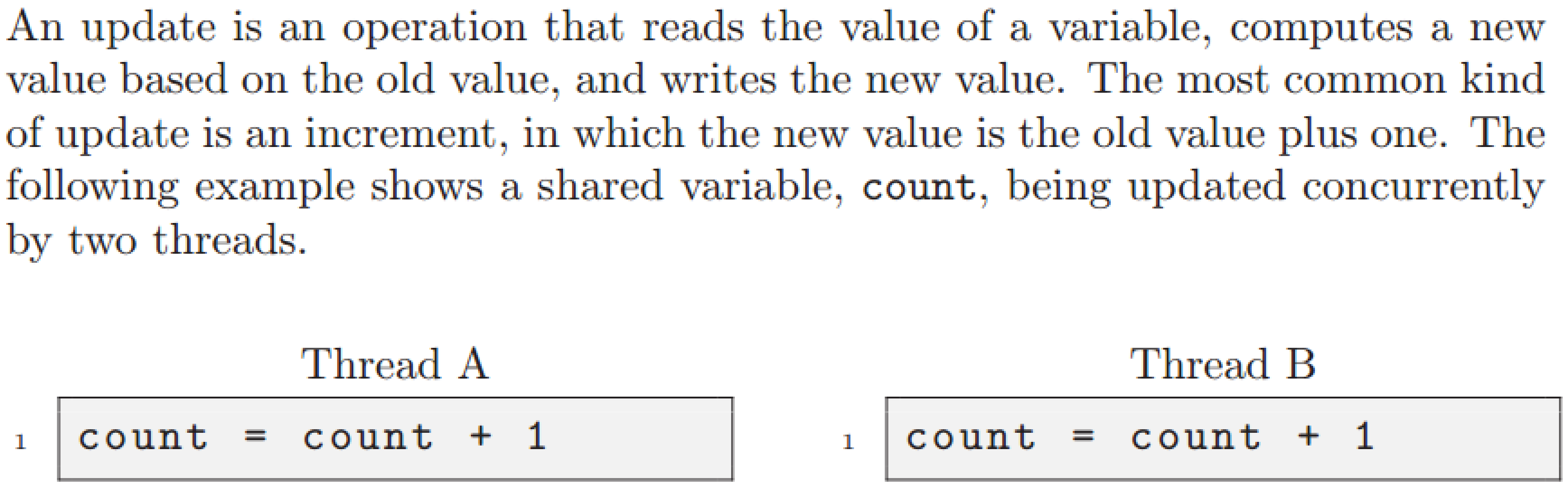
Thread B

```
1  x = 7
```

What value of **x** gets printed? What is the final value of **x** when all these statements have executed? It depends on the order in which the statements are executed, called the **execution path**. One possible path is $a1 < a2 < b1$, in which case the output of the program is 5, but the final value is 7.

Разделяемая память

2) Конкурентное обновление



An update is an operation that reads the value of a variable, computes a new value based on the old value, and writes the new value. The most common kind of update is an increment, in which the new value is the old value plus one. The following example shows a shared variable, `count`, being updated concurrently by two threads.

Thread A

```
count = count + 1
```

Thread B

```
count = count + 1
```

Разделяемая память

Thread A

```
temp = count
count = temp + 1
```

Thread B

```
temp = count
count = temp + 1
```

This kind of problem is subtle because it is not always possible to tell, looking at a high-level program, which operations are performed in a single step and which can be interrupted. In fact, some computers provide an increment instruction that is implemented in hardware and cannot be interrupted. An operation that cannot be interrupted is said to be atomic.

Разделяемая память

```
// Поток 1:
while (!stop) {
  x++;
  …
}
```

```
// Поток 2:
while (!stop) {
  if (x%2 == 0)
    System.out.println("x=" + x);
  …
}
```

Пусть x=0. Предположим, что выполнение программы происходит в таком порядке:

1. Оператор if в потоке 2 проверяет *x* на чётность.
2. Оператор «*x*++» в потоке 1 увеличивает *x* на единицу.
3. Оператор вывода в потоке 2 выводит «*x*=*1*», хотя, казалось бы, переменная проверена на чётность.

Паттерны синхронизации

- Блокировка
- Сигнализирование
- Установка барьера

Примитивы синхронизации

- Семафоры
- Мьютексы
- Условные переменные
- Барьеры

Классический семафор

1. When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are allowed to perform are increment (increase by one) and decrement (decrease by one). You cannot read the current value of the semaphore.

2. When a thread decrements the semaphore, if the result is negative, the thread blocks itself and cannot continue until another thread increments the semaphore.

3. When a thread increments the semaphore, if there are other threads waiting, one of the waiting threads gets unblocked.

To say that a thread blocks itself (or simply “blocks”) is to say that it notifies the scheduler that it cannot proceed. The scheduler will prevent the thread from running until an event occurs that causes the thread to become unblocked. In the tradition of mixed metaphors in computer science, unblocking is often called “waking”.

Блокировка

Thread A

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Thread B

```
mutex.wait()
    # critical section
    count = count + 1
mutex.signal()
```

Блокировка

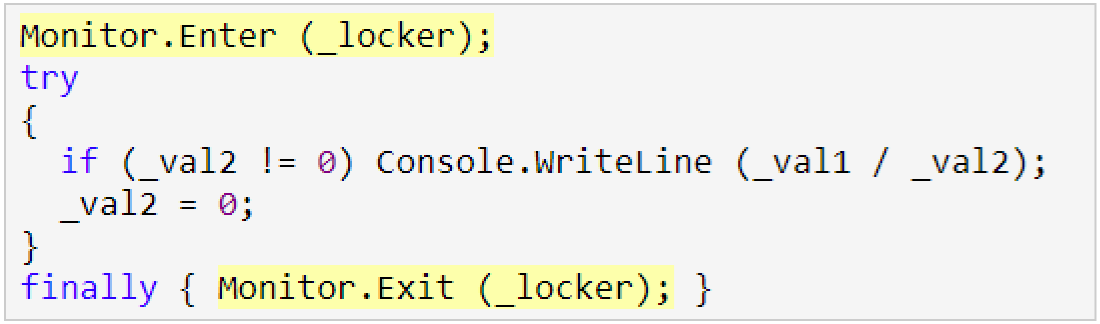
```
static SemaphoreSlim _sem = new SemaphoreSlim (1);    // Capacity of 1
static int _count;

static void Main()
{
  for (int i = 0; i < 5; i++) new Thread(Foo).Start();
}

static void Foo()
{
    _sem.Wait();
    _count++;
    _sem.Release();
}
```

SemaphoreSlim (1)   =>   Mutex

Монитор

A monitor is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. Monitors also have a mechanism for signaling other threads that their condition has been met.

1. It is the collection of condition variables and procedures combined together in a special kind of module or a package.
2. The processes running outside the monitor can’ t access the internal variable of monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

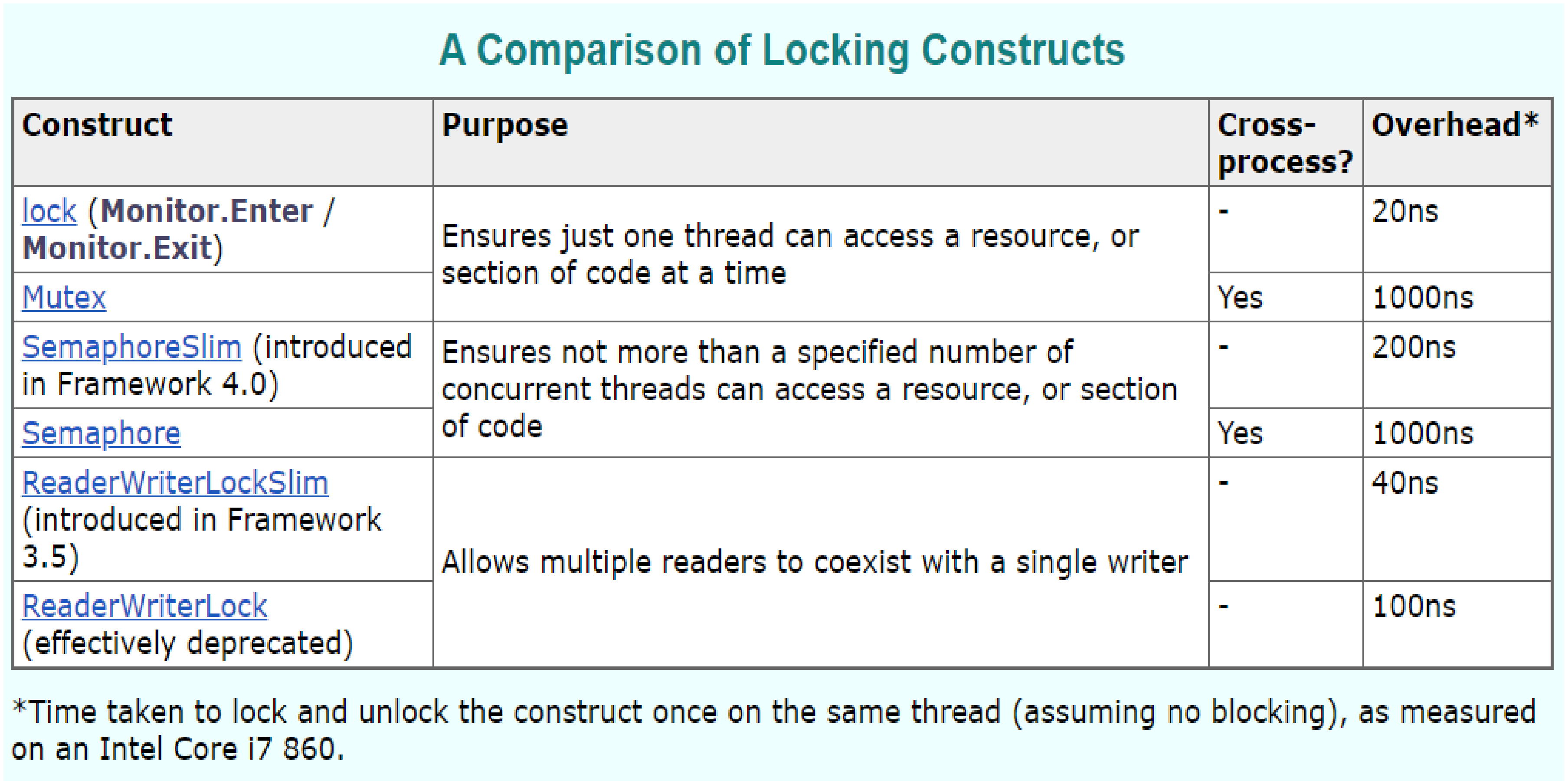


```
Monitor.Enter (_locker);
try
{
  if (_val2 != 0) Console.WriteLine (_val1 / _val2);
  _val2 = 0;
}
finally { Monitor.Exit (_locker); }
```

Инструкция lock

```
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1, _val2;

  static void Go()
  {
    lock (_locker)
    {
      if (_val2 != 0) Console.WriteLine (_val1 / _val2);
      _val2 = 0;
    }
  }
}
```

Блокирующие примитивы



A Comparison of Locking Constructs

Construct	**Purpose**	**Cross-process?**	**Overhead***
lock (**Monitor.Enter** / **Monitor.Exit**)	Ensures just one thread can access a resource, or section of code at a time	-	20ns
Mutex		Yes	1000ns
SemaphoreSlim (introduced in Framework 4.0)	Ensures not more than a specified number of concurrent threads can access a resource, or section of code	-	200ns
Semaphore		Yes	1000ns
ReaderWriterLockSlim (introduced in Framework 3.5)	Allows multiple readers to coexist with a single writer	-	40ns
ReaderWriterLock (effectively deprecated)		-	100ns

*Time taken to lock and unlock the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

Сигнализирование

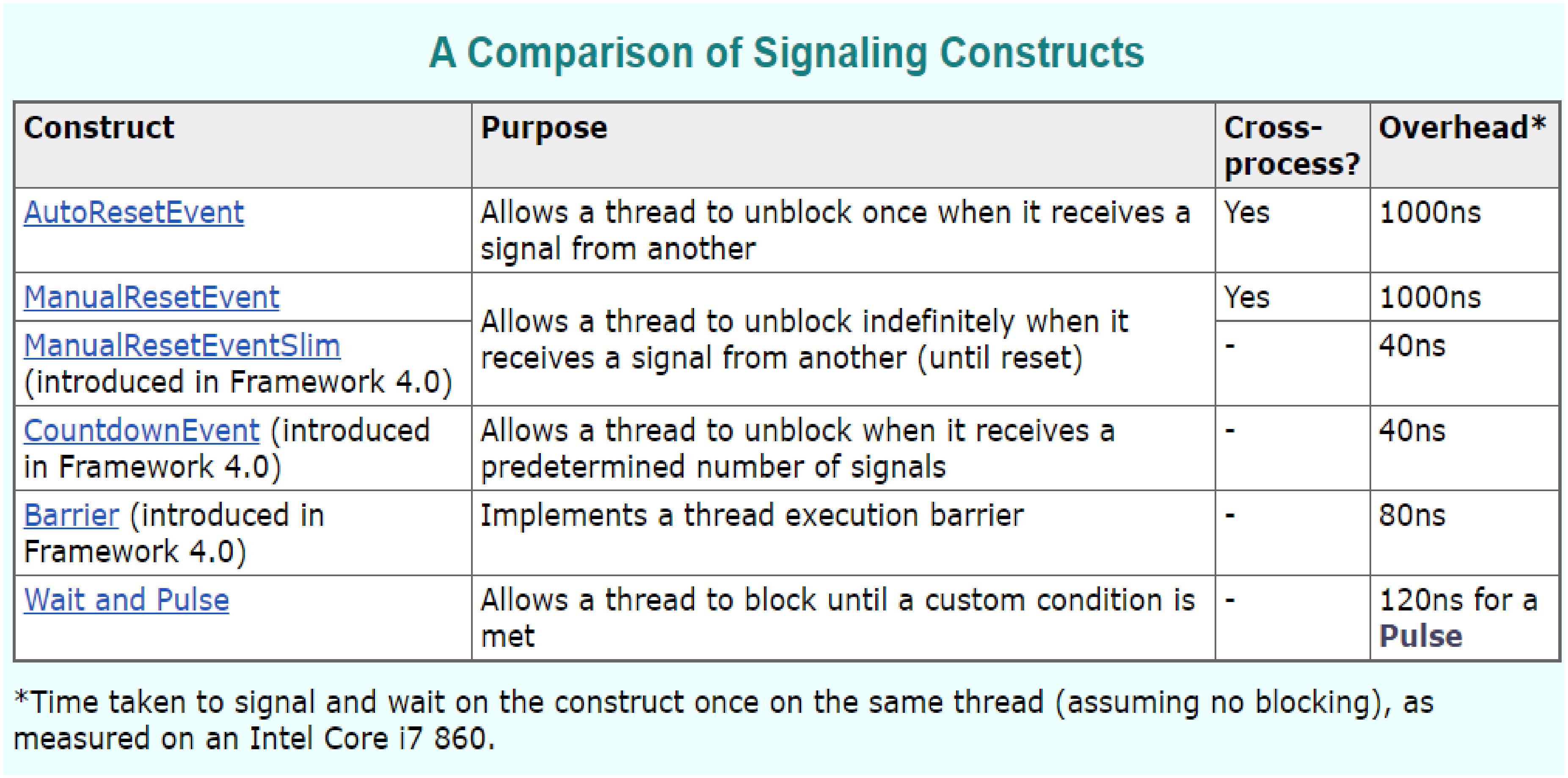
Thread A

```
statement a1
sem.signal()
```

Thread B

```
sem.wait()
statement b1
```

Сигнализирование



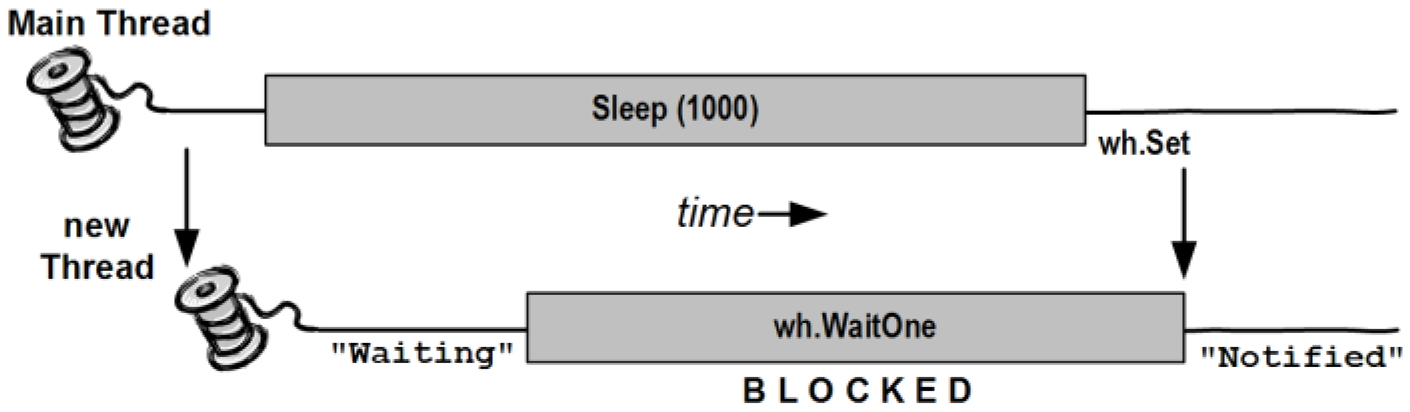
A Comparison of Signaling Constructs

Construct	Purpose	Cross-process?	Overhead*
AutoResetEvent	Allows a thread to unblock once when it receives a signal from another	Yes	1000ns
ManualResetEvent	Allows a thread to unblock indefinitely when it receives a signal from another (until reset)	Yes	1000ns
ManualResetEventSlim (introduced in Framework 4.0)		-	40ns
CountdownEvent (introduced in Framework 4.0)	Allows a thread to unblock when it receives a predetermined number of signals	-	40ns
Barrier (introduced in Framework 4.0)	Implements a thread execution barrier	-	80ns
Wait and Pulse	Allows a thread to block until a custom condition is met	-	120ns for a **Pulse**

*Time taken to signal and wait on the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

AutoResetEvent / ManualResetEvent

```
static EventWaitHandle _waitHandle = new AutoResetEvent (false);

 static void Main()
 {
   new Thread (Waiter).Start();
   Thread.Sleep (1000);                    // Pause for a second...
   _waitHandle.Set();                      // Wake up the Waiter.
 }

 static void Waiter()
 {
   Console.WriteLine ("Waiting...");
   _waitHandle.WaitOne();                  // Wait for notification
   Console.WriteLine ("Notified");
 }
```



Pulse / Wait / WaitAll

Барьер (рандеву)

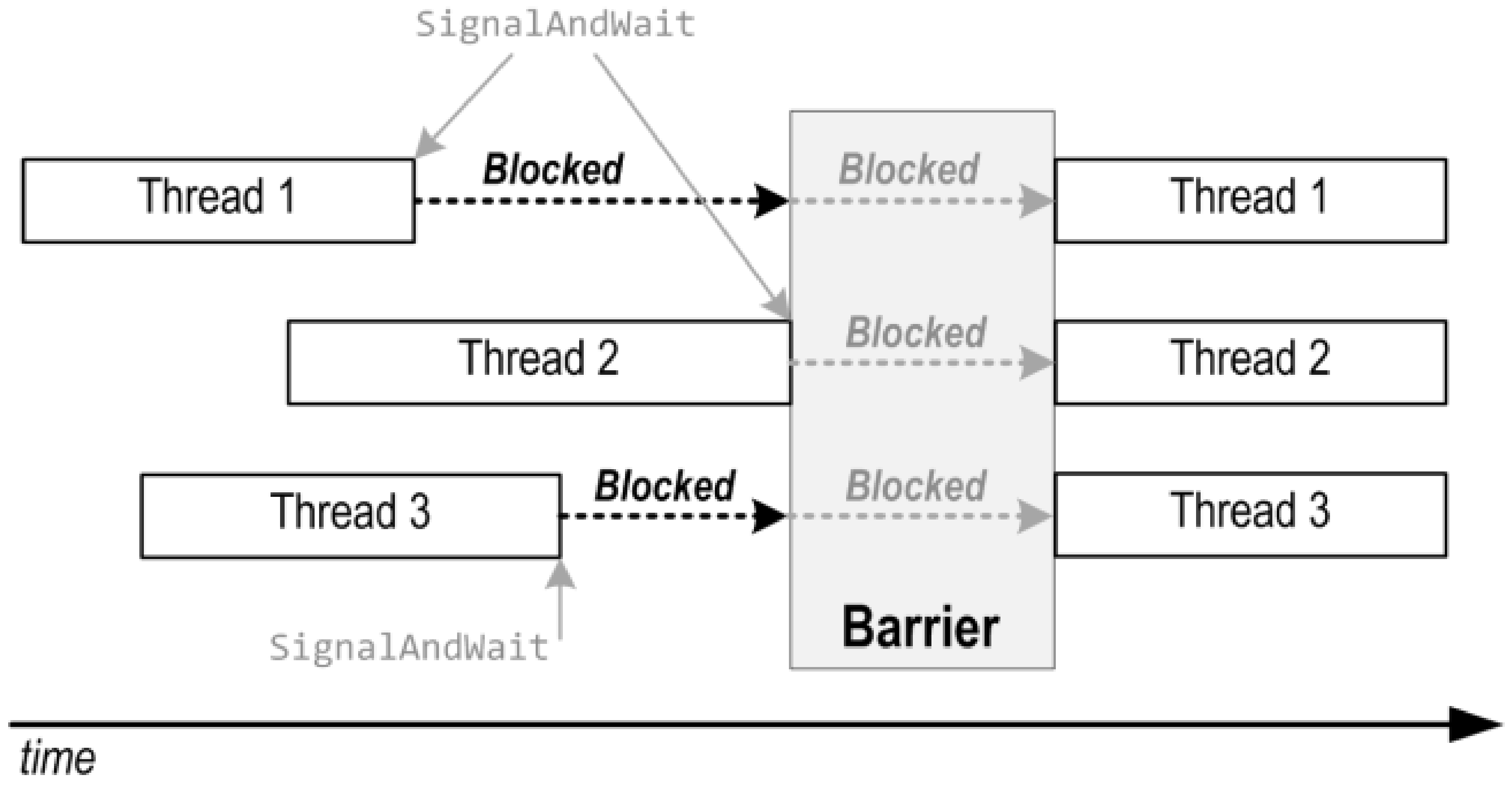
Thread A

```
statement a1
aArrived.signal()
bArrived.wait()
statement a2
```

Thread B

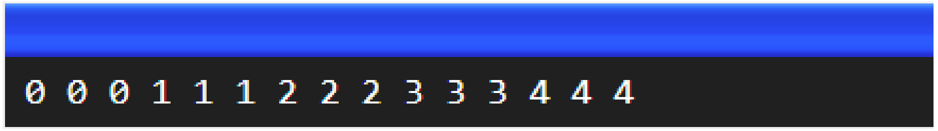
```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

Барьер (рандеву)



Барьер (рандеву)

```
static Barrier _barrier = new Barrier (3);

static void Main()
{
  new Thread (Speak).Start();
  new Thread (Speak).Start();
  new Thread (Speak).Start();
}

static void Speak()
{
  for (int i = 0; i < 5; i++)
  {
    Console.Write (i + " ");
    _barrier.SignalAndWait();
  }
}
```



```
0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

Синхронизация без блокировок

- Lock-free
- Wait-free
- Obstruction-free

https://sites.google.com/site/1024cores/home/in-russian/lock--wait--obstruction--atomic-free-algorithms

Неблокирующая синхронизация: Interlocked

- Interlocked.Increment
- Interlocked.Decrement
- Interlocked. Exchange
- Interlocked.CompareExchange

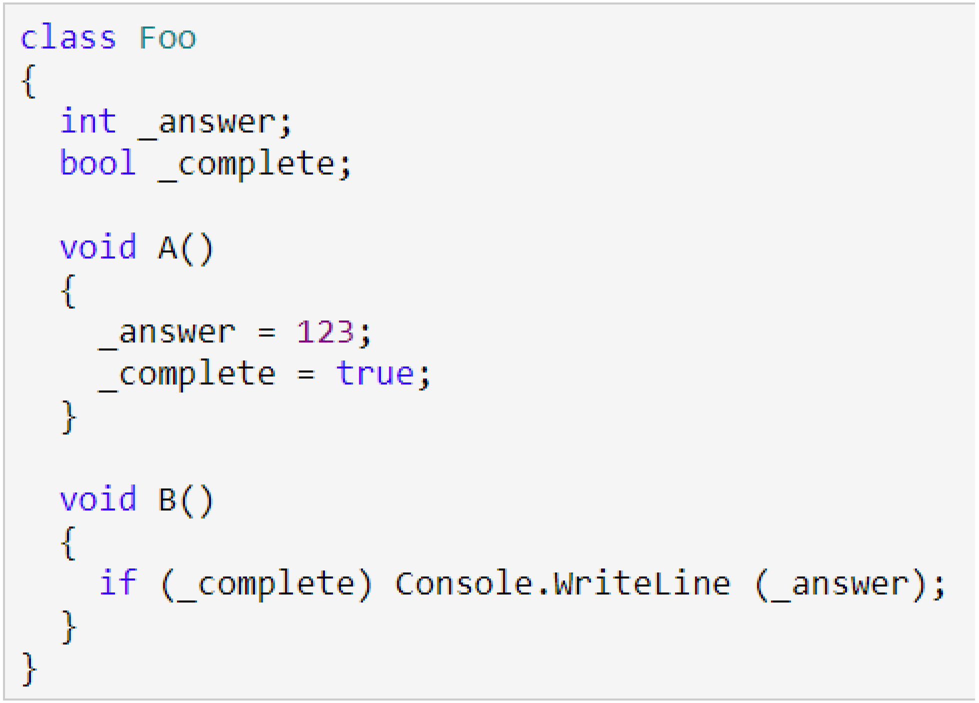
```
int newValue = Interlocked.Increment(ref value);
int setValue = Interlocked.Exchange(ref value, 73); // assign
```

Неблокирующая синхронизация: MemoryBarrier

If methods A and B ran concurrently on different threads, might it be possible for B to write "0" ? The answer is yes — for the following reasons:

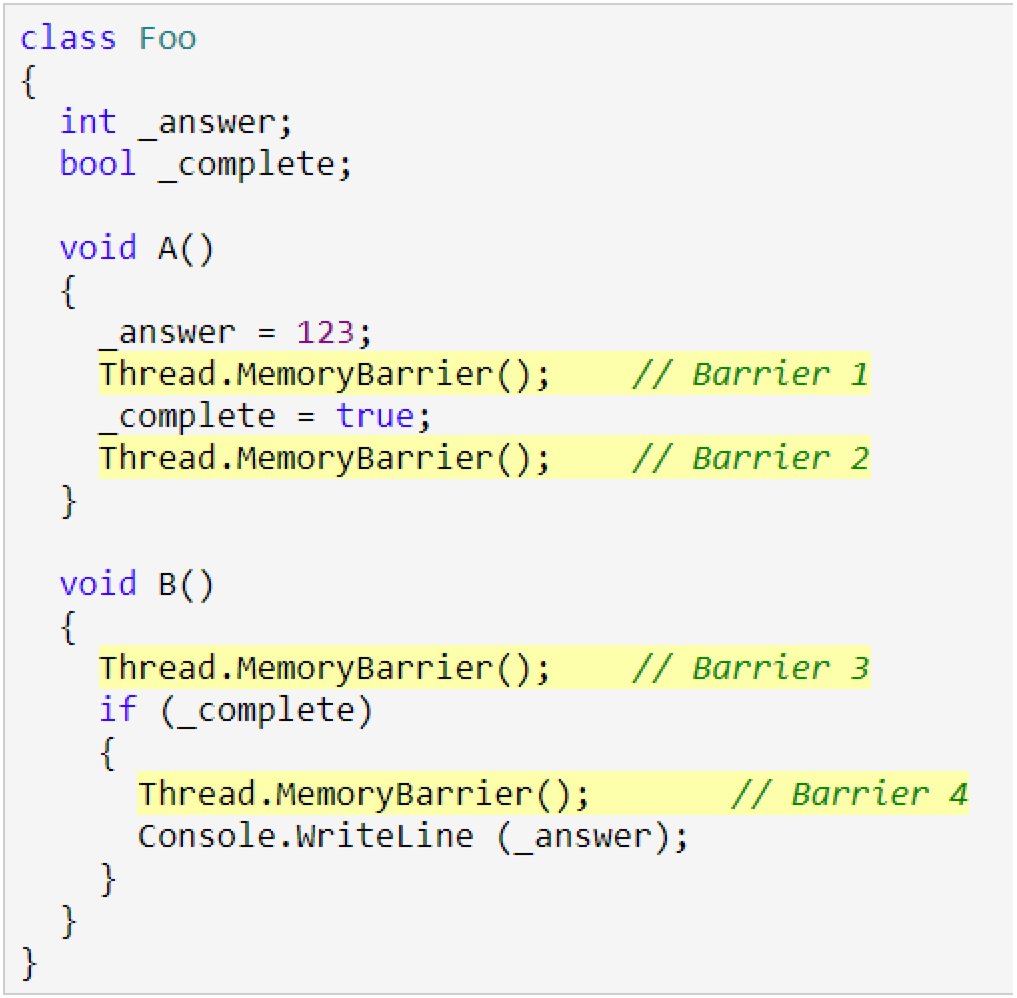
•The compiler, CLR, or CPU may *reorder* your program's instructions to improve efficiency.

•The compiler, CLR, or CPU may introduce caching optimizations such that assignments to variables won't be visible to other threads right away.



```
class Foo
{
  int _answer;
  bool _complete;

  void A()
  {
    _answer = 123;
    _complete = true;
  }

  void B()
  {
    if (_complete) Console.WriteLine (_answer);
  }
}
```

Неблокирующая синхронизация: MemoryBarrier



```
class Foo
{
  int _answer;
  bool _complete;

  void A()
  {
    _answer = 123;
    Thread.MemoryBarrier();    // Barrier 1
    _complete = true;
    Thread.MemoryBarrier();    // Barrier 2
  }

  void B()
  {
    Thread.MemoryBarrier();    // Barrier 3
    if (_complete)
    {
      Thread.MemoryBarrier();       // Barrier 4
      Console.WriteLine (_answer);
    }
  }
}
```

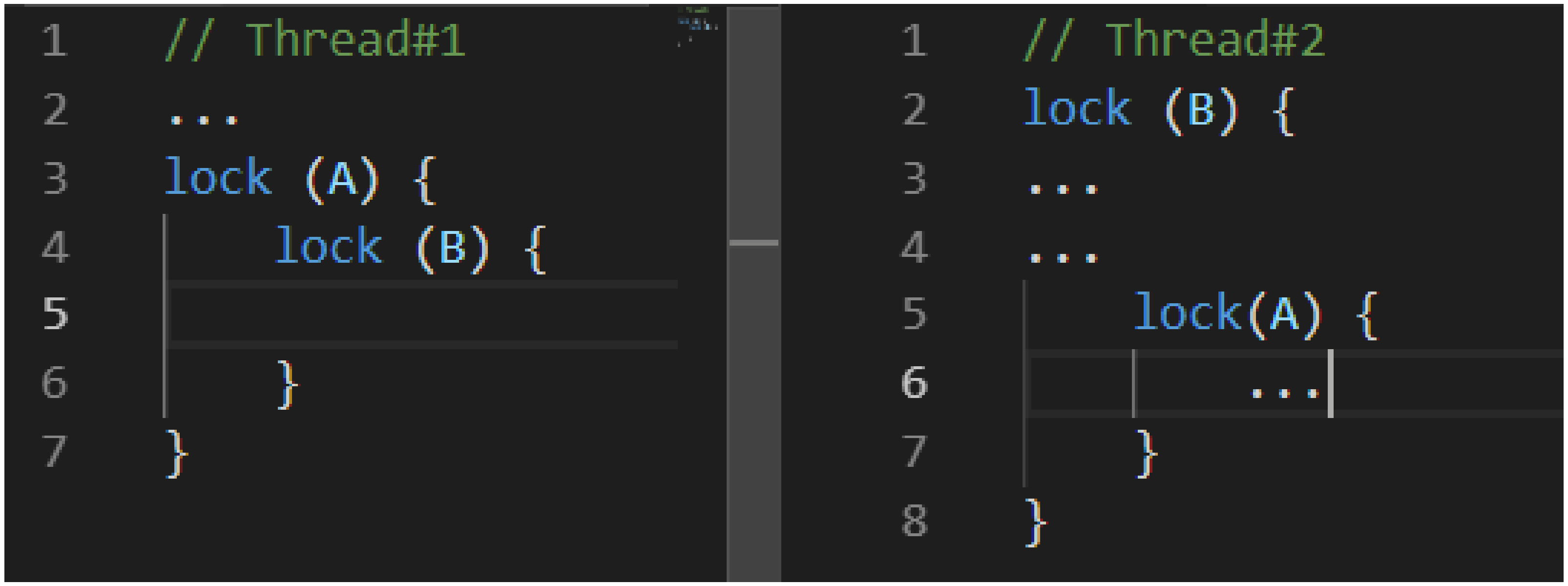
Проблемы синхронизации

-Race condition (non-determinism)

-Deadlock

- Starvation

Самый частый deadlock



```
// Thread#1
...
lock (A) {
    lock (B) {

    }
}
```

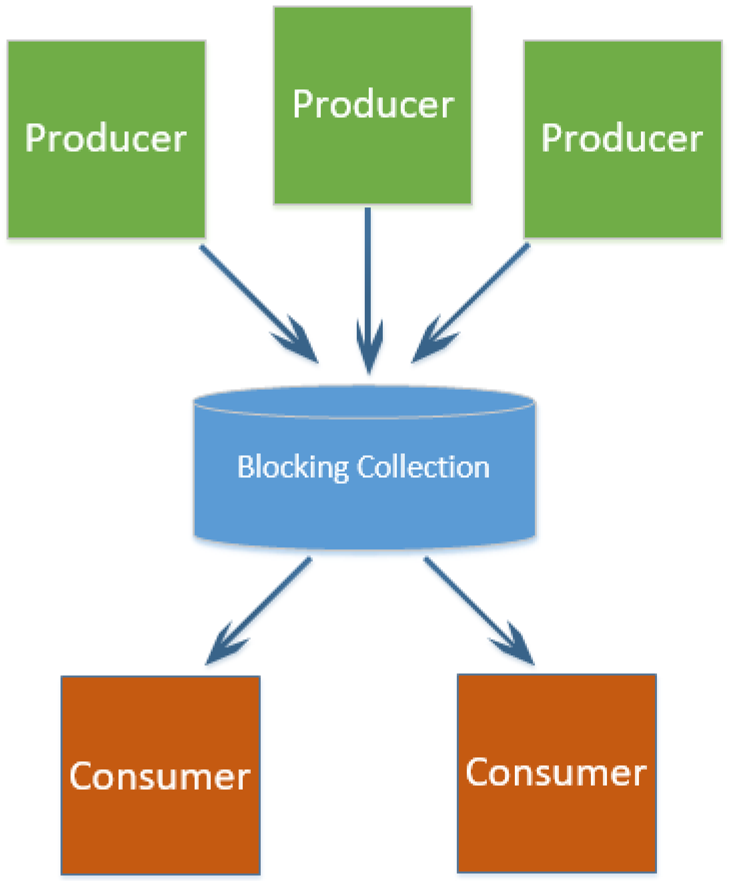
```
// Thread#2
lock (B) {
...
...
    lock(A) {
        ...
    }
}
```

https://habr.com/ru/post/459514/

Классические задачи синхронизации

- Producer / Consumer
- Dining philosophers
- Readers / Writers
- Sleeping barber

Producer / Consumer



Producer / Consumer

```
q = Queue()

# это мютекс для печати (не относится к паттерну)
print_lock = threading.Lock()


def consumer():
    while True:
        # достаем из очереди задачу
        worker = q.get()
        # делаем что надо
        time.sleep(1)
        with print_lock:
            print('Поток {} обрабатывает задачу {}'.format(
                    threading.current_thread().name, worker))
        # отмечаем, что задача отработана
        q.task_done()

def producer(n):
    # каждый продюсер кладет в очередь 5 задач
    for task in range(5):
        q.put(n*5 + task)
```

```
# создаем 10 консьюмеров
for _ in range(10):
    t = threading.Thread(target=consumer,
daemon=True)
    t.start()

# создаем 4 продюсеров
for i in range(4):
    t = threading.Thread(target=producer,
daemon=True, args=(i,))
    t.start()

# ждем завершения потока
q.join()
```

Producer / Consumer

Here’ s how the producer/consumer queue works:

1. A queue is set up to describe work items - or data upon which work is performed.
2. When a task needs executing, it’ s enqueued, allowing the caller to get on with other things.
3. One or more worker threads plug away in the background, picking off and executing queued items.

The advantage of this model is that you have precise control over how many worker threads execute at once. This can allow you to limit consumption of not only CPU time, but other resources as well. If the tasks perform intensive disk I/O, for instance, you might have just one worker thread to avoid starving the operating system and other applications. Another type of application may have 20. You can also dynamically add and remove workers throughout the queue’ s life.

A producer/consumer queue typically holds items of data upon which (the same) task is performed. For example, the items of data may be filenames, and the task might be to encrypt those files.

P/C queue (ver.1)

```
class ProducerConsumerQueue : IDisposable
{
  EventWaitHandle _wh = new AutoResetEvent (false);
  Thread _worker;
  readonly object _locker = new object();
  Queue<string> _tasks = new Queue<string>();

  public ProducerConsumerQueue()
  {
    _worker = new Thread (Work);
    _worker.Start();
  }

  public void EnqueueTask (string task)
  {
    lock (_locker) _tasks.Enqueue (task);
    _wh.Set();
  }

  public void Dispose()
  {
    EnqueueTask (null); // Signal the consumer to exit
    _worker.Join();     // Wait for the consumer's thread to finish
    _wh.Close();        // Release any OS resources
  }
```

P/C queue (ver.1)

```
static void Main()
{
  using (ProducerConsumerQueue q = new
                             ProducerConsumerQueue())
  {
    q.EnqueueTask ("Hello");
    for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
    q.EnqueueTask ("Goodbye!");
  }
}
```

```
void Work()
{
    while (true)
    {
      string task = null;
      lock (_locker)
        if (_tasks.Count > 0)
        {
          task = _tasks.Dequeue();
          if (task == null) return;
        }
      if (task != null)
      {
        Console.WriteLine ("Performing task: " + task);
        Thread.Sleep (1000);  // simulate work...
      }
      else
        _wh.WaitOne(); // No more tasks - wait for a signal
    }
  }
}
```

P/C queue (ver.2)

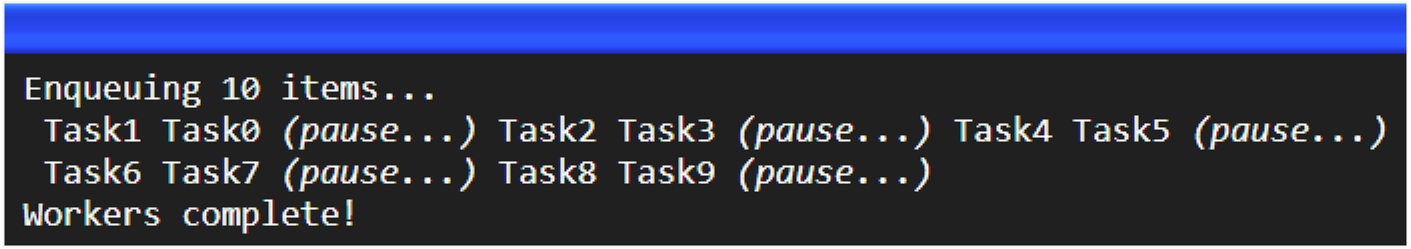
```
public class PCQueue
{
  readonly object _locker = new object();
  Thread[] _workers;
  Queue<Action> _itemQ = new Queue<Action>();

  public PCQueue (int workerCount)
  {
    _workers = new Thread [workerCount];

    // Create and start a separate thread for each worker
    for (int i = 0; i < workerCount; i++)
      (_workers [i] = new Thread (Consume)).Start();
  }

  public void Shutdown (bool waitForWorkers)
  {
    // Enqueue one null item per worker to make each exit.
    foreach (Thread worker in _workers)
      EnqueueItem (null);

    // Wait for workers to finish
    if (waitForWorkers)
      foreach (Thread worker in _workers)
        worker.Join();
  }
```

P/C queue (ver.2)

```
public void EnqueueItem (Action item)
  {
    lock (_locker)
    {
      _itemQ.Enqueue (item);           // We must pulse because we're
      Monitor.Pulse (_locker);         // changing a blocking condition.
    }
  }

  void Consume()
  {
    while (true)                        // Keep consuming until
    {                                   // told otherwise.
      Action item;
      lock (_locker)
      {
        while (_itemQ.Count == 0) Monitor.Wait (_locker);
        item = _itemQ.Dequeue();
      }
      if (item == null) return;         // This signals our exit.
      item();                           // Execute item.
    }
  }
```

P/C queue (ver.2)

```
static void Main()
{
  PCQueue q = new PCQueue (2);

  Console.WriteLine ("Enqueuing 10 items...");

  for (int i = 0; i < 10; i++)
  {
    int itemNumber = i;      // To avoid the captured variable trap
    q.EnqueueItem (() =>
    {
      Thread.Sleep (1000);    // Simulate time-consuming work
      Console.Write (" Task" + itemNumber);
    });
  }

  q.Shutdown (true);
  Console.WriteLine();
  Console.WriteLine ("Workers complete!");
}
```



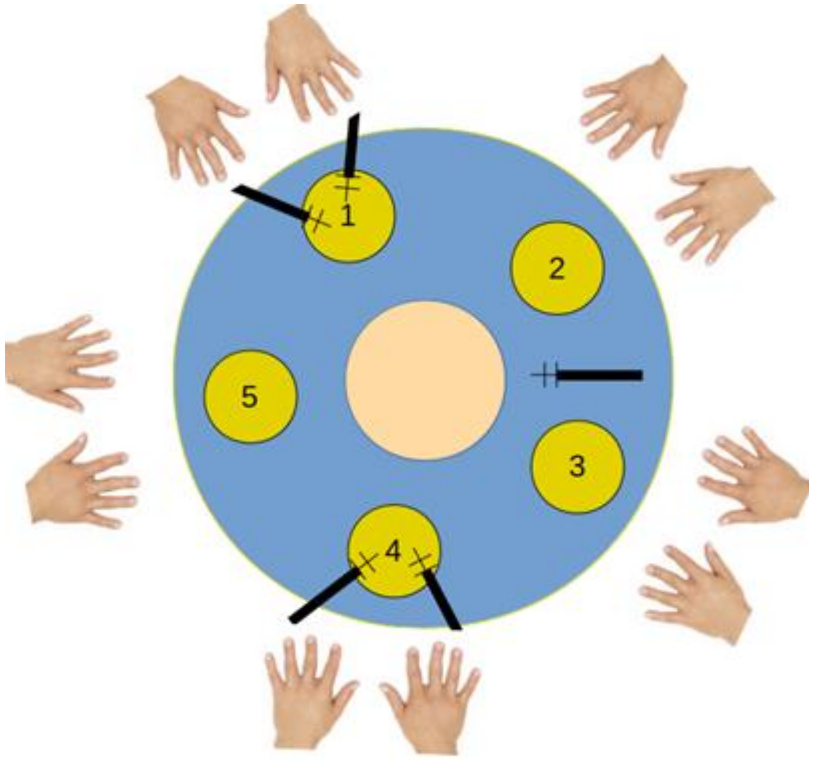
```
Enqueuing 10 items...
 Task1 Task0 (pause...) Task2 Task3 (pause...) Task4 Task5 (pause...)
 Task6 Task7 (pause...) Task8 Task9 (pause...)
Workers complete!
```

P/C with BlockingCollection

```
private static BlockingCollection<int> data = new BlockingCollection<int>();

private static void Producer()
{
    for (int ctr = 0; ctr < 10; ctr++)
    {
        data.Add(ctr);
        Thread.Sleep(100);
    }
}

private static void Consumer()
{
    foreach (var item in data.GetConsumingEnumerable())
    {
        Console.WriteLine(item);
    }
}

static void Main(string[] args)
{
    var producer = Task.Factory.StartNew(() => Producer());
    var consumer = Task.Factory.StartNew(() => Consumer());
    Console.Read();
}
```

Dining Philosophers

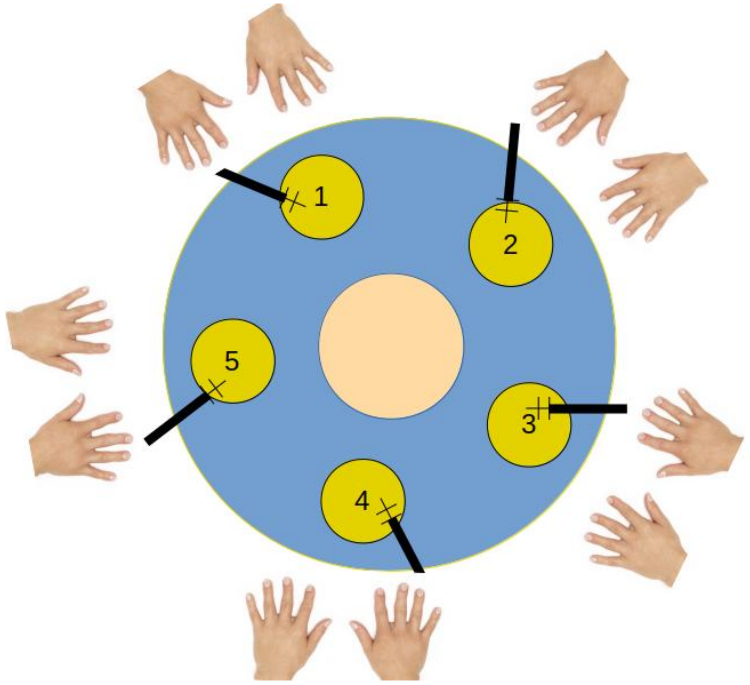
The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosophers and 5 spoons/chopsticks. A philosopher needs both their right and a left spoon to eat. A hungry philosopher may only eat if there are both spoons available. Otherwise, a philosopher puts down their spoon and begin thinking again.



Наивное решение

```
Semaphore spoon[5] = { 1, 1, 1, 1, 1 }
// ...

// thread function:

while (true)
{
        // thinking

        spoon[x].wait();               // wait for left chop
        spoon[x+1]%5].wait();          // wait for right chop

         // eating

        spoon[x+1]%5].signal();        // release right chop
        spoon[x].signal();             // release left chop

         // finish eating
}
```

Возможен дедлок



Решения без дедлока

- Монитор
- Иерархия ресурсов
- «Официант»
- «Один чудак»

«Один чудак»

```
Semaphore spoon[5] = { 1, 1, 1, 1, 1 }
// …

// thread function for #0,1,2,3:

while (true)
{
        // thinking

        spoon[x].wait();
        spoon[x+1]%5].wait();

         // eating

        spoon[x+1]%5].signal();
        spoon[x].signal();

         // finish eating
}
```

```
// thread function for #4:

while (true)
{
        // thinking

        spoon[x+1]%5].wait();
        spoon[x].wait();

         // eating

        spoon[x].signal();
        spoon[x+1]%5].signal();

         // finish eating
}
```

Монитор

https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/

Sleeping barber

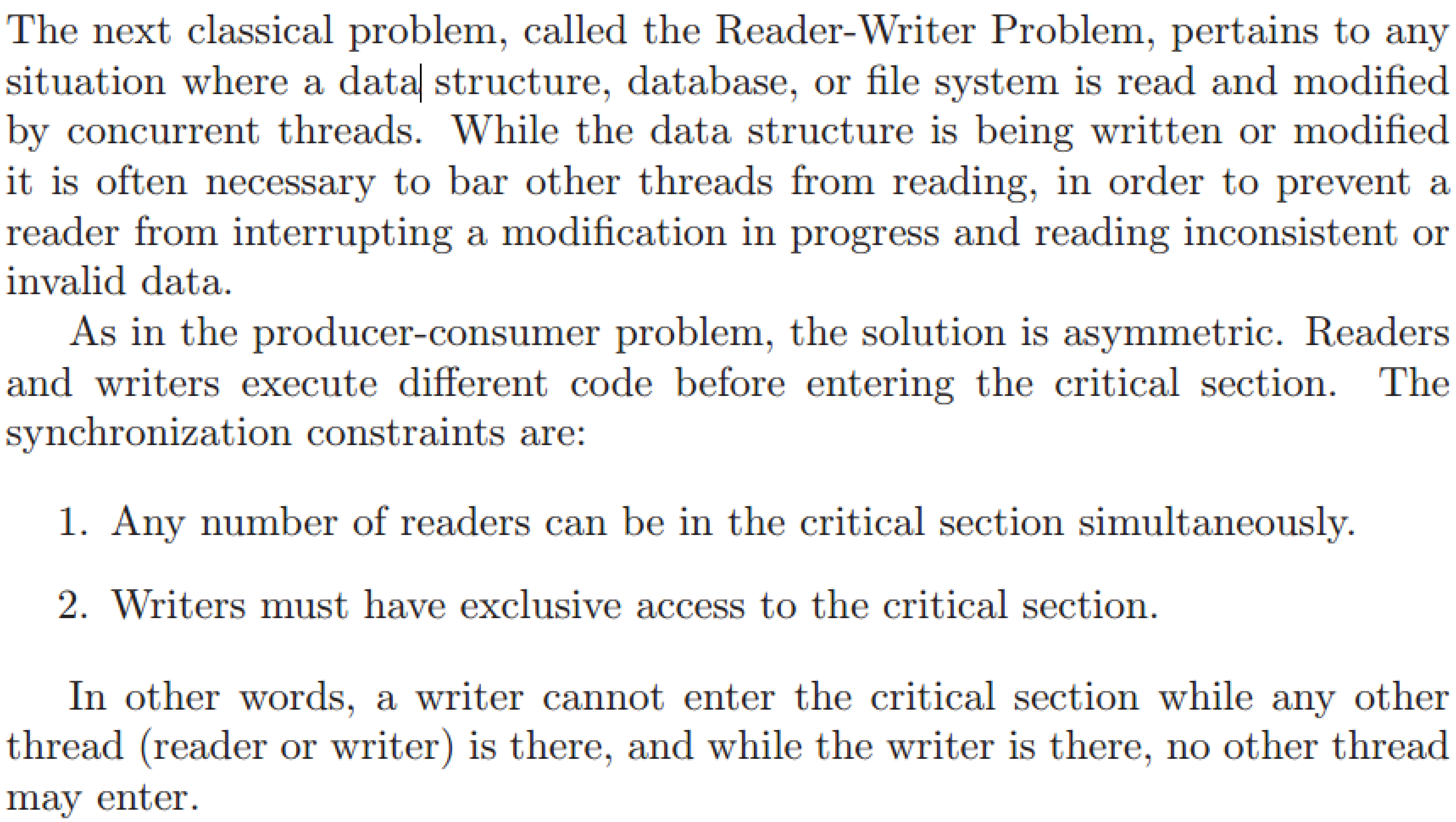
The barber has one barber's chair in a cutting room and a waiting room containing a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts their hair. If there are none, he returns to the chair and sleeps in it.

Each customer, when they arrive, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits their turn. If there is no free chair, the customer leaves.

Решение (без дедлока, но starvation возможен)

```
Semaphore barberReady = 0
Semaphore accessWRSeats = 1     # if 1, the number of seats in the waiting room can be incremented or decremented
Semaphore custReady = 0         # the number of customers currently in the waiting room, ready to be served
int numberOfFreeWRSeats = N     # total number of seats in the waiting room

def Barber():
  while true:                   # Run in an infinite loop.
    wait(custReady)             # Try to acquire a customer - if none is available, go to sleep.
    wait(accessWRSeats)         # Awake - try to get access to modify # of available seats, otherwise sleep.
    numberOfFreeWRSeats += 1    # One waiting room chair becomes free.
    signal(barberReady)         # I am ready to cut.
    signal(accessWRSeats)       # Don't need the lock on the chairs anymore.
    # (Cut hair here.)

def Customer():
  while true:                   # Run in an infinite loop to simulate multiple customers.
    wait(accessWRSeats)         # Try to get access to the waiting room chairs.
    if numberOfFreeWRSeats > 0: # If there are any free seats:
      numberOfFreeWRSeats -= 1  #   sit down in a chair
      signal(custReady)         #   notify the barber, who's waiting until there is a customer
      signal(accessWRSeats)     #   don't need to lock the chairs anymore
      wait(barberReady)         #   wait until the barber is ready
      # (Have hair cut here.)
    else:                       # otherwise, there are no free seats; tough luck --
      signal(accessWRSeats)     #   but don't forget to release the lock on the seats!
      # (Leave without a haircut.)
```

Readers / Writers



The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

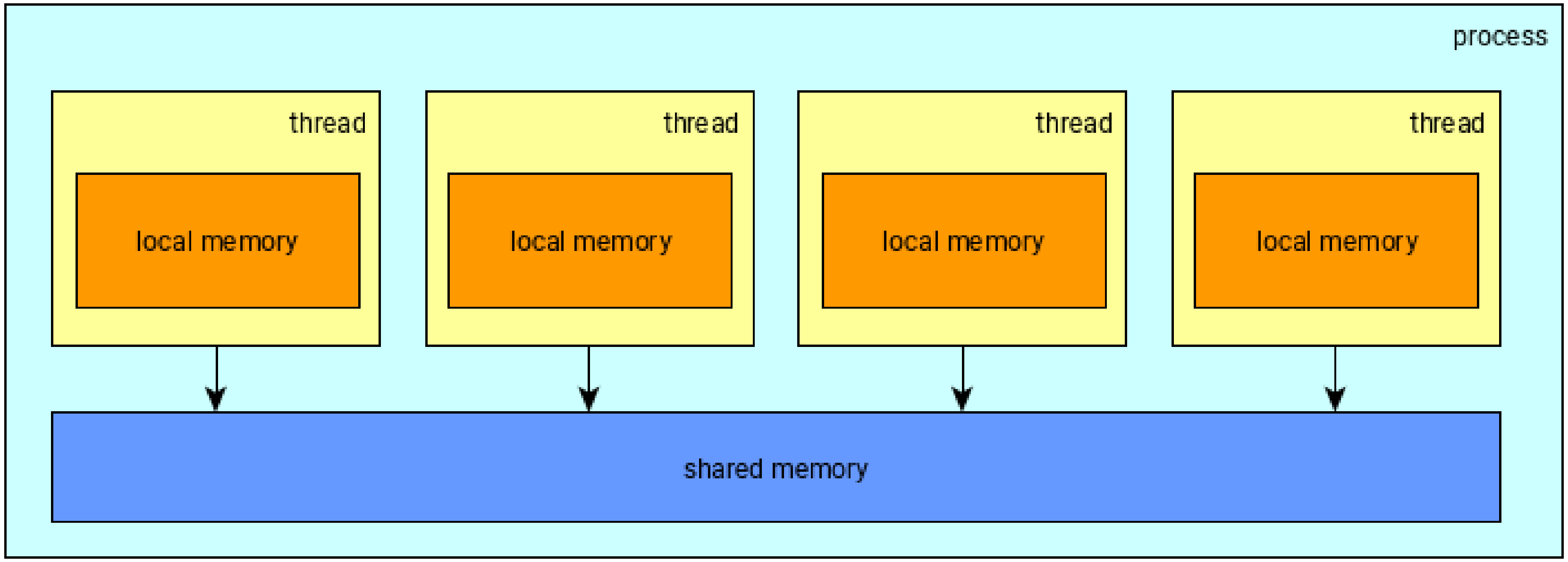
1. Any number of readers can be in the critical section simultaneously.

2. Writers must have exclusive access to the critical section.

In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

OpenMP

Библиотека OpenMP подходит только для программирования систем с общей памятью, при этом используется параллелизм потоков. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса.



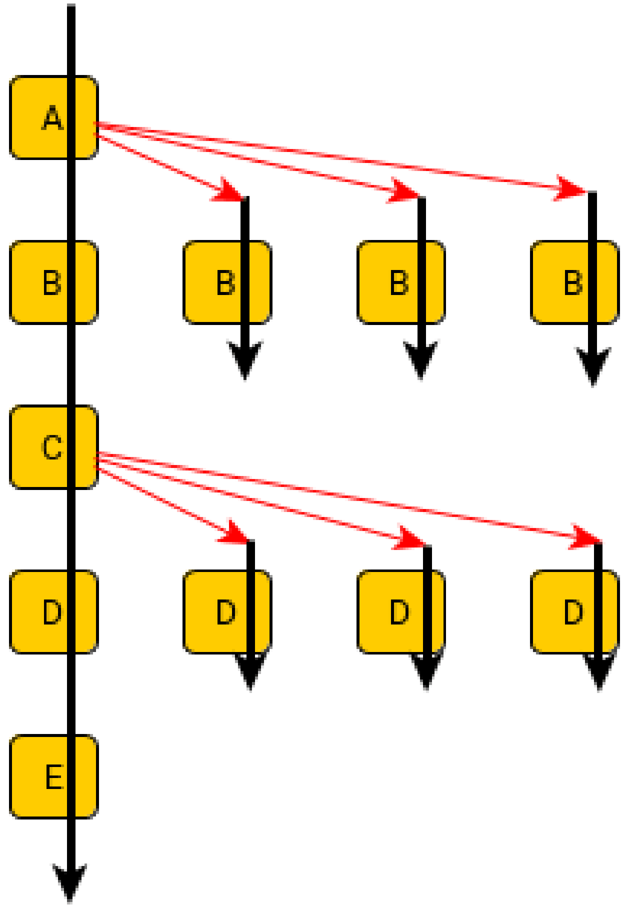
OpenMP

Распараллеливание в OpenMP выполняется явно при помощи вставки в текст программы специальных директив, а также вызова вспомогательных функций. При использовании OpenMP предполагается *SPMD*-модель *(Single Program Multiple Data)* параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код.

Интерфейс OpenMP задуман как стандарт для программирования на масштабируемых SMP-системах (SSMP, ccNUMA и других) в модели общей памяти (*shared memory model*). В стандарт OpenMP входят спецификации набора директив компилятора, вспомогательных функций и переменных среды. OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор «подчиненных» (slave) потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами, причём количество процессоров не обязательно должно быть больше или равно количеству потоков.

OpenMP

```
#include "omp.h"

int main() {
   // A - single thread
#pragma omp parallel
   {
     // B - many threads
   }
   // C - single thread
#pragma omp parallel
   {
     // D - many threads
   }
   // E - single thread
}
```



OpenMP

Далее примеры из источника:

https://www.ibm.com/developerworks/ru/library/au-aix-openmp-framework/index.html

OpenMP

Visual Studio:
/openmp

Qt:
QMAKE_CXXFLAGS+= -fopenmp
QMAKE_LFLAGS += -fopenmp

```
#include <iostream>

int main()
{
    #pragma omp parallel
    {
        std::cout << "Hello World!\n";
    }
}
```

```
user$ g++ test1.cpp –fopenmp
user$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Intel i7 Core
(4 физ.ядра, 2 лог.ядра)

OpenMP num_threads

1

```
int main()
{
  #pragma omp parallel num_threads(5)
  {
     std::cout << "Hello World!\n";
  }
}
```

2

```
#include <omp.h>

int main()
{
  omp_set_num_threads(5);

  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

3

```
user$ export OMP_NUM_THREADS=6
user$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

OpenMP   parallel for

```
int main( )
{
    int a[1000000], b[1000000];
    // ... код для инициализации массивов a и b;
    int c[1000000];

    #pragma omp parallel for
    for (int i = 0; i < 1000000; ++i)
        c[i] = a[i] * b[i] + a[i-1] * b[i+1];

    // ... выполняем некоторые действия с массивом c
 }
```

OpenMP omp_get_wtime

```
int main(int argc, char *argv[])
{
    int i, nthreads;
    clock_t clock_timer;
    double wall_timer;
    double c[1000000];
    for (nthreads = 1; nthreads <=8; ++nthreads)
    {
        clock_timer = clock();
        wall_timer = omp_get_wtime();

        #pragma omp parallel for private(i) num_threads(nthreads)
        for (i = 0; i < 1000000; i++)
          c[i] = sqrt(i * 4 + i * 2 + i);

        std::cout << "threads: " << nthreads <<  " time on clock(): " <<
            (double) (clock() - clock_timer) / CLOCKS_PER_SEC
           << " time on wall: " <<  omp_get_wtime() - wall_timer << "\n";
    }
}
```

Результаты

```
threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
threads: 5 time on clock(): 0.01594 time on wall: 0.00359988
threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
threads: 8 time on clock(): 0.01678 time on wall: 0.00237703
```

OpenMP critical (section)

```
#pragma omp critical (section1)
{
     myhashtable.insert("key1", "value1");
}

#pragma omp critical (section1)
{
     myhashtable.insert("key2", "value2");
}
```

Мьютексы в OpenMP

•omp_init_lock: эта API-функция должна использоваться первой при обращении к типу omp_lock_t и предназначена для инициализации. Необходимо отметить, что сразу после инициализации блокировка будет находиться в исходном (unset) состоянии.

•omp_destroy_lock: уничтожает блокировку. В момент вызова этой API-функции блокировка должна находиться в исходном состоянии; это означает, что нельзя вызвать функцию omp_set_lock, а затем уничтожить блокировку.

•omp_set_lock: устанавливает omp_lock_t, т. е. активирует мьютекс. Если поток не может установить блокировку, он продолжает ожидать до тех пор, пока такая возможность не появится.

•omp_test_lock: пытается установить блокировку, если она доступна; возвращает 1 в случае успеха и 0 в случае неудачи. Это функция является *неблокирующей*, т. е. она не заставляет поток ожидать установления блокировки.

•omp_unset_lock: сбрасывает блокировку в исходное состояние.

Пример потокобезопасной реализации очереди

```
class omp_q : public myqueue<int> {
public:
   typedef myqueue<int> base;
   omp_q( ) {
      omp_init_lock(&lock);
   }
   ~omp_q() {
       omp_destroy_lock(&lock);
   }
   bool push(const int& value) {
      omp_set_lock(&lock);
      bool result = this->base::push(value);
      omp_unset_lock(&lock);
      return result;
   }
   bool trypush(const int& value)
   {
       bool result = omp_test_lock(&lock);
       if (result) {
          result = result && this->base::push(value);
          omp_unset_lock(&lock);
      }
      return result;
   }
   // likewise for pop
private:
   omp_lock_t lock;
};
```

OpenMP parallel sections

```
int main( )
{
 #pragma omp parallel
 {
   cout << "Это выполняется во всех потоках\n";
   #pragma omp sections
   {
     #pragma omp section
     {
       cout << "Это выполняется параллельно\n";
     }
     #pragma omp section
     {
       cout << "Последовательный оператор 1\n";
       cout << "Это всегда выполняется после оператора 1\n";
     }
     #pragma omp section
     {
       cout << "Это тоже выполняется параллельно\n";
     }
   }
 }
}
```

Результат

```
user$ ./a.out
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется параллельно
Последовательный оператор 1
Это тоже выполняется параллельно
Это всегда выполняется после оператора 1
```

Инициализация private

idx не инициализируется 100:

```
int idx = 100;
#pragma omp parallel private(idx)
{
  printf("В потоке %d idx = %d\n", omp_get_thread_num(), idx);
}
```

idx инициализируется 100:

```
int idx = 100;
#pragma omp parallel  firstprivate(idx)
{
  printf("В потоке %d idx = %d\n", omp_get_thread_num(), idx);
}
```

Инициализация private

```
int main()
{
  int idx = 100;
  int main_var = 2120;

  #pragma omp parallel for private(idx)  lastprivate(main_var)
  for (idx = 0; idx < 12; ++idx)
  {
    main_var = idx * idx;
    printf("В потоке %d idx = %d main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("Возврат в главный поток со значением переменной main_var = %d\n", main_var);
}
```

Отладка многопоточного кода

https://docs.microsoft.com/en-us/visualstudio/debugger/get-started-debugging-multithreaded-apps?view=vs-2019