Работа с потоками в C#

С помощью объектов System.Threading.Thread

Основные методы:

```
Start()     – запуск,
Interrupt() – приостановка выполнения,
Abort()     – прерывание выполнения,
Join()      – ожидание завершения.
```

События и исключения

обрабатываются в том же потоке, в котором они возникают.

Создание и завершение дополнительного потока для «службы»

```
public partial class DataAnalysisSrv : ServiceBase
    {
        Thread Mt; //объявление
        protected override void OnStart(string[] args)
        {//стандартный обработчик запуска службы
            try
            {
                Mt = new Thread(StartMonitor); //создание
                Mt.Start(); //запуск
            }
            catch (Exception err)
            {
                 ... //сообщить об ошибке запуска
            }
        }
        protected override void OnStop()
        {//стандартный обработчик остановки службы
                Mt.Abort(); //прервать выполнение
                Mt.Join();  //дождаться завершения
                 ...        //сообщить об успешном завершении работы
        }
        private void StartMonitor()
        {
            ...//выполнение работы
        }
```

Вызов метода `Abort`

приводит к появлению в потоке исключения `ThreadAbortException`, которое должно быть обработано.

Обработка прерывания потока

```
private void StartMonitor()
        {
            try
            {
                ... //выполнение
            }
            catch (ThreadAbortException ab)
            {
                //получена команда завершения
                  ... //освободить занятые ресурсы
                  ... //(закрыть файлы, соединения с БД и т.д.)
                return;
            }
            catch (Exception err)
            {
                  ... //обработать возможные ошибки
                return;
            }
            finally
            {
                  ... //освободить занятые ресурсы
            }
        }
```

Фоновые потоки (свойство **IsBackground=Истина**)

Когда все основные потоки завершились, среда завершает процесс несмотря на наличие фоновых потоков.

```
t = new Thread(...);
t.IsBackground = true;
t.Start();
```

Средства синхронизации

Оператор `lock` – простейший способ создания критической секции.

Класс `Monitor`

методы:

`Enter`, `TryEnter` – захватить объект;

`Wait` – освободить объект и ждать сигнала;

`Pulse`, `PulseAll` – послать сигнал;

`Exit` – освободить объект.

Классы `Semaphore`, `Mutex`, **структура** `SpinLock`

Создание критической секции - оператор lock

```
public partial class DataAnalysisSrv : ServiceBase
    {
        Thread Mt;
        static object Locker = new object();
...
    }
...

                lock (Locker)
                {
                    //работа с общими ресурсами
                }
...
```

Потокобезопасные коллекции

`System.Collections.Concurrent`

ConcurrentDictionary,
ConcurrentStack,
ConcurrentQueue.

Доступ к элементам формы из другого потока

Если есть уверенность в отсутствии конфликтов, то можно ***отключить проверку***.

Иначе

Использовать метод `Invoke(Delegate, Object[])`класса `Control`. Он Выполняет указанный делегат в том потоке, которому принадлежит основной дескриптор окна элемента управления, с указанным списком аргументов.

Доступ к элементам формы из другого потока

```
private void button_get_stat_Click(object sender, EventArgs e)
        {
            ...
        //запрет проверки
            Control.CheckForIllegalCrossThreadCalls = false;
                GetStatThread = new Thread(GetStat);
                GetStatThread.Start();
            ...
        }
private void GetStat()
        {
            ...
        //добавление точки на диаграмму
            chart_stat.Series[0].Points.AddXY(x, y);
            ...
        }
```

Безопасный доступ к элементам формы

```
 private void ThreadProcSafe() //будет запущена в другом потоке
        {
            this.SetText("This text was set safely.");
        }
 //обработчик нажатия кнопки
 private void setTextSafeBtn_Click(object sender, EventArgs e)
        {
            this.demoThread = new Thread(new ThreadStart(this.ThreadProcSafe));
            this.demoThread.Start();
        }
delegate void SetTextCallback(string text);
        private void SetText(string text)
        {
            if (this.textBox1.InvokeRequired)//если выполняется в другом потоке
            {
                SetTextCallback d = new SetTextCallback(SetText);
//запускает в том потоке которому принадлежит базовый дескриптор окна
                this.Invoke(d, new object[] { text });
            }
            else
            {
                this.textBox1.Text = text;
            }
        }
```

Что нежелательно делать
при изучении параллельного программирования.
Task, async и await

Task, async и await

- *высокоуровневый механизм, позволяющий пользователю (программисту) описать алгоритм специальным образом (в терминах задач) и самостоятельно принимающий решение об использовании потоков.*

Цитата msdn.microsoft.com

«работа, выполняемая объектом Task обычно выполняется асинхронно в пуле потоков, а не синхронно в потоке основного приложения»

…

«***Асинхронный метод не выполняется в собственном потоке. Метод выполняется в текущем контексте синхронизации***».

Особенности синхронизации потоков на «низком» уровне

Не работает и не должно:

```
char mut1 = 0;

// функция использующая общие ресурсы
void func() {

      while (mut1) {}

      mut1 = 1;
      // код с использованием общих ресурсов
      mut1 = 0;
}
```

Простой и удобный пример:

```
int c = 0;
timespec t = { 0, 0 };
void * thread_function(void *x)
{
        for (int i = 0; i < 1000000; i++, c++)
                pthread_delay_np(&t);
        return 0;
}
int _tmain(int argc, _TCHAR* argv[])
{
        pthread_t T1, T2;
        pthread_create(&T1, NULL, thread_function, NULL);
        pthread_create(&T2, NULL, thread_function, NULL);
        pthread_join(T1, NULL);
        pthread_join(T2, NULL);
        printf("c= %d\n", c);
        system("pause");
        return 0;
}
```

Осторожно оптимизация!

Алгоритм Деккера

(должно, но не работает :(

```
flag[0] := false
flag[1] := false
turn := 0 // or 1
```

Thread 0:

```
flag[0] := true
while flag[1] = true {
 if turn = 1 {
  flag[0] := false
  while turn = 1 {}
  flag[0] := true } }
// к.с.
...
turn := 1
flag[0] := false
// конец к.с.
...
```

Thread 1:

```
flag[1] := true
while flag[0] = true {
 if turn = 0 {
  flag[1] := false
  while turn = 0 {}
  flag[1] := true } }
// к.с.
...
turn := 0
flag[1] := false
// конец к.с.
...
```

Причины неработоспособности алгоритма *Деккера* на современных многопроцессорных системах.

1. *Уже упомянутая оптимизация*.
2. Реализация не учитывает вопросов своевременной актуализации (когерентности) кэш-памяти процессоров. ***Иными словами изменение сделанное одним ядром становится известно другому в неопределенный момент времени***.

Атомарные объекты

– объекты **операции** над которыми **являются неделимыми** по отношению к другим операциям, использующим тот же объект (*они не могут быть прерваны другими, а результат не может быть получен, до их окончания*).

Атомарные объекты в C++11

1. `std::atomic <T>`
2. `std::atomic_flag`

T – *некоторый тип данных.*

std::atomic_flag

два метода:

test_and_set, clear

ПРИМЕР:

```
class my_spinlock_t
{
private:
	std::atomic_flag Locked;
public:
	my_spinlock_t(){ Locked.clear(); }

	void lock(){ while (Locked.test_and_set());}

	void unlock() { Locked.clear(); }
};
```

std::atomic <T>

Содержит следующие методы.
is_lock_free – предоставляет информации о свободен ли объект данного типа от блокировок.

store – Помещает новое значение в объект.
load – Извлекает значение из объекта.
exchange – Заменяет значение в объекте на новое и возвращает старое.

operator=() и **operator T()** эквивалентны **store** и **load.**
И другие.

Пример:

```
class my_spinlock_t
{
private:
	std::atomic<bool> Locked;
public:
	my_spinlock_t(){ Locked = false; }

	void lock() { while (Locked.exchange(true)); }

	void unlock() { Locked.store(false); }
};
```

Если задача действительно состоит в корректном увеличении счетчика, его достаточно сделать атомарным:

```
std::atomic<int> c = 0;

timespec t = { 0, 0 };
void * thread_function(void *x)
{
      for (int i = 0; i < 1000000; i++, c++)
            pthread_delay_np(&t);
      return 0;
}
```

Предсказуемый порядок *чтения/модификации объекта различными потоками существует* ***только для атомарных*** *объектов.*

Memory ordering

Любая операция над содержимым атомарного объекта принимает одним из параметров `memory_order`, по умолчанию это всегда `std::memory_order_seq_cst`

```
enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

Последовательная согласованность (sequential consistency) `memory_order_seq_cst`

гарантирует синхронизацию операций `store` и `load` в различных потоках.

Пример:

```
std::atomic<bool> flag = false;
int v = 0;
...
```

```
//Thread 0:
v = 2;//NA
flag.store(true);//A
...
```

```
//Thread 1:
while (!flag.load())
...
if(v == 2)//B
...
```

Запись в `v` не может «перепрыгнуть» `store`,

и чтение `v` не может «выпрыгнуть» за `load`.

1. ПС операции (**<u>любые</u>**) не могут выполняться полностью одновременно. Они обязаны завершаться в разное время (*как будто существует единый глобальный поток*).

2. Чтение гарантированно получает последнее записанное значение.

```
std::atomic_int integer{0};
std::atomic_bool flagA{false};
std::atomic_bool flagB{false};

void thread1()
{
  flagB.store(true);
  if(flagA.load())
    ++integer;
}
 void thread2()
{
  flagA.store(true);
  if(flagB.load())
    ++integer;
}

int main()
{
  std::thread firstThread(&thread1);
  std::thread secondThread(&thread2);
  firstThread.join();
  secondThread.join();
  assert(integer > 0);
```

integer всегда > 0

`memory_order_relaxed`

Минимальные гарантии:

Запись и чтение атомарного объекта являются атомарными операциями.

Если поток прочитал значение **B**, которое было записано в атомарный объект, то при следующем чтении этого же объекта, в этом потоке, гарантировано будет прочитано либо это же **B**, либо более позднее значение **L**, но никогда не будет прочитано значение, которое было записано ранее **B**.

Пример:

```
std::atomic<bool> flag = false;
int v = 0;
...
```

```
//Thread 0:
v = 2;//NA
flag.store(true,
std::memory_order_relaxed);
//A
...
```

```
//Thread 1:
while
(!flag.load(std::memory_
order_relaxed))
...
if(v == 2)//B
...
```

Нет гарантий, что v будет = 2.

Модель захвата-освобождения

Синхронизируются операции захвата / освобождения

`memory_order_acquire` – операции использующие данный маркер являются операциями захвата (чтение).

`memory_order_release` –” – операциями освобождения (запись).

`memory_order_acq_rel` –” – операциями захвата для предыдущих операций и операциями освобождения для последующих операций.

1. Операции **над различными объектами** **могут** выполняться полностью одновременно.

2. Порядок чтения аналогичен модели `relaxed`.

3. Модель позволяет сформировать отношение ***происходит до*** для не атомарных операций.

```
std::atomic_int integer{0};
std::atomic_bool flagA{false};
std::atomic_bool flagB{false};

void thread1()
{
  flagB.store(true, std::memory_order_release);//№П1-1
  if(flagA.load(std::memory_order_acquire))//№П2-2
    ++integer;//№П2-3
}
 void thread2()
{
  flagA.store(true, std::memory_order_release);//№П2-1
  if(flagB.load(std::memory_order_acquire))//№П2-2
    ++integer;//№П2-3
}

int main()
{
  std::thread firstThread(&thread1);
  std::thread secondThread(&thread2);
  firstThread.join();
  secondThread.join();
  assert(integer > 0);
```

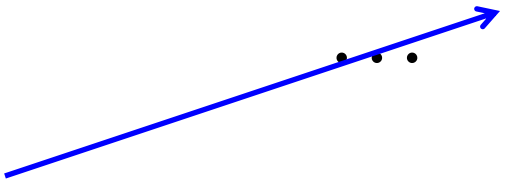
Нет гарантий, что integer > 0.

Пример:

```
std::atomic<bool> flag = false;
int v = 0;
...
```

```
//Thread 0:
v = 2;//NA
flag.store(true,
std::memory_order_release);
//A
...
```

```
//Thread 1:
while
(!flag.load(std::memory_
order_acquire))
...
if(v == 2)//B
...
```



Всегда истина

Модель «потребления»

`memory_order_consume` – операции использующие данный маркер являются операциями «потребления».

`memory_order_release` –” – операциями освобождения (операции записи).

Модель формирует более слабые отношения, предполагающие явные информационные зависимости.

```
int integer = 0;
char character ='A';
std::atomic<int*> atomicPtr{nullptr};
//...

void thread1()
{
  character = 'Z';
  integer = 55;
  atomicPtr.store(&integer,
std::memory_order_release);
}

void thread2()
{
  int* ptr = nullptr;
  while((ptr=atomicPtr.load(std::memory_order_c
onsume)) == nullptr)
    ;
  assert(*ptr == 55);
  assert(character == 'Z');
}
```


Гарантии есть

Гарантий нет

Источники

cppreference.com

https://ru.cppreference.com/w/cpp/atomic/memory_order
https://en.cppreference.com/w/cpp/atomic

Серия статей на **scrutator.me**

1. «Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый».
2. «Добро пожаловать в параллельный мир. Часть 4: Порядки и отношения».
3. «Добро пожаловать в параллельный мир. Часть 5: Граница на замке».