C#
-CAHPTER10-

SOUL SEEK

목차

THREAD&TASK

1. THREAD & TASK

프로세스

실행파일이 실행되어 메모리에 적재된 인스턴스
프로그램의 실행파일을 실행시키는 역할을 한다.
반드시 하나이상의 **Thead**로 구성된다.

Thread

운영체제가 **CPU**시간을 할장하는 기본단위
프로세스를 구성하는 요소 ➔ 프로세스는 굵은 밧줄, **Thread**는 밧줄을 구성하는 실.

1. THREAD & TASK

멀티 스레드의 장점

- 사용자 대화형 프로그램(콘솔 프로그램과 **GUI** 프로그램 모두)에서 멀티 스레드를 이용하면 응답성을 높일 수 있다.
  ➔파일을 전송 중일 때 다른 행동이 가능
- 멀티 프로세스 방식에 비해 멀티 스레드 방식이 자원 공유가 쉽다는 점이 있다.
  ➔**GUI**가 없는 웹 서버 같은 서버용 애플리케이션에서 많이 취하는 구조.
  ➔쓰레드 끼리 코드내의 변수를 같이 사용하는 것만으로도 데이터를 쉽게 교환 할 수 있다.
  →레이드를 할때 보스의 **HP**에 대한 데이터들은 각각 유저의 공격을 했고 이런 결과가 나왔다는 값을 확인 해 볼 수 있다.
- 프로세스를 띄우기 위해 메모리와 자원을 할당하는 작업은 비용이 비싼 편이지만, 쓰레드를 띄울 때는 이미 프로세스에 할당된 메모리 자원을 그대로 사용하므로 메모리와 자원을 할당하는 비용을 지불하지 않아도 된다.

프로세스와 쓰레드의 데이터 교환 방식의 차이

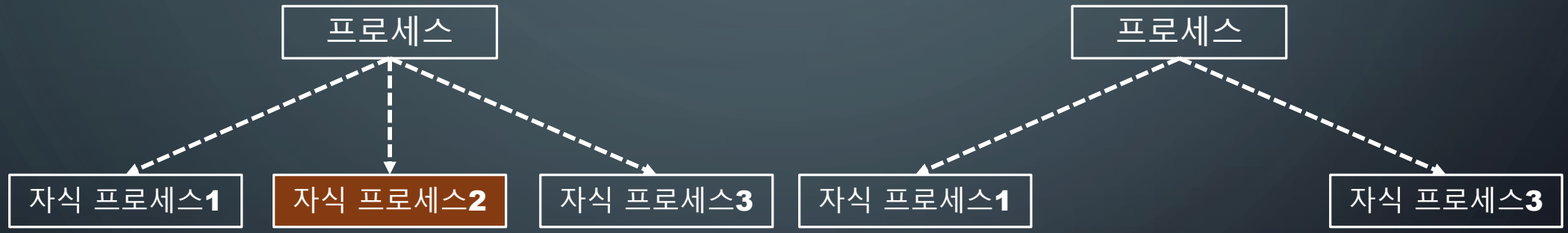
IPC를 통한 프로세스 간의 데이터 교환

변수를 이용한 스레드 간의 데이터 교환

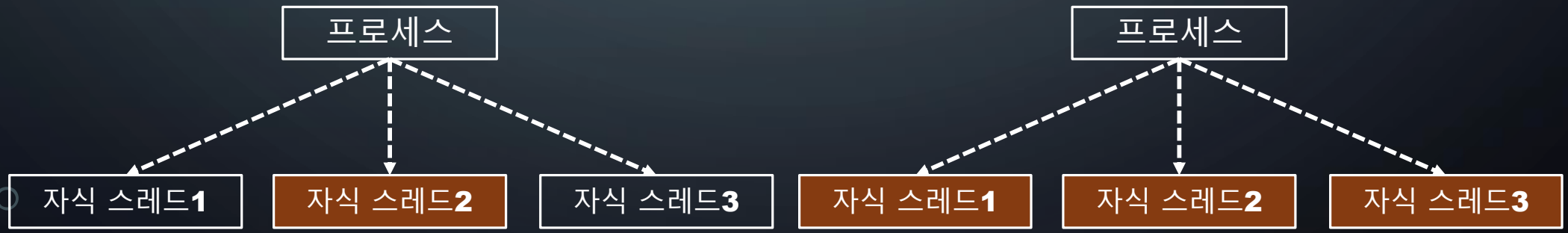
1. THREAD & TASK

멀티 스레드 단점

- 멀티 스레드 구조의 소프트웨어의 구현이 매우 까다롭고 테스트 역시 쉽지 않다.
- 멀티 프로세스 기반의 소프트웨어는 여러 개의 자식 프로세스 중 하나에 문제가 생기면 그 자식 프로세스 하나가 죽는 것 이상으로 영향이 확산되지 않지만, 멀티 스레드는 기반의 소프트웨어에서는 자식 스레드 중 한에 문제가 생기면 전체 프로세스에 영향을 받게 된다.
- 너무 과다한 스레드의 사용은 오히려 성능을 저하시킨다.
  ➔스레드가 **CPU**를 사용하기 위해서는 작업 간 전환**(Context Swithing)**을 해야 하는데, 이 작업 간 전환이 많은 비용을 소모한다.
  ➔스레드를 잘 사용하기 위해서는 경험이 반드시 필요하다 그렇기 때문에 적어도 어떻게 사용 할 수 있는지는 알아둬야 한다.



멀티 프로세스 구조에서 자식 프로세스에 문제가 생긴 경우



멀티 스레드 구조에서 자식 스레드에 문제가 생긴 경우

1. THREAD & TASK

멀티 스레드 단점

- **System.Threading.Thread**를 제공한다.
- **Thread** 클래스 사용방법은..
  1. **Thread**의 인스턴스를 생성한다, 이 때 생성자의 매개 변수로 스레드가 실행할 메소드를 매개 변수로 넘긴다.
  2. **Thread.Start()** 메소드를 호출하여 스레드를 시작한다.
  3. **Thread.Join()** 메소드를 호출하여 스레드가 끝날 때까지 기다린다.

```
static void DoSomting()                                    // 스레드가 실행할 메소드
{
    for(int i = 0; i < 5; i++)
    {
        Console.WriteLine("DoSomthing : {0}", i);
    }
}

static void Main()
{
    Thread t1 = new Thread(new ThreadStart(DoSomething)); // Thread의 인스턴스 생성
    t1.Start();    // 스레드 시작
    t1.Join();     // 스레드의 종료 대기 ➔ 메인 스레드로 다시 합류한다는 의미 이기 때문에 Join
}
```

1. THREAD & TASK

스레드 임의 종료 시키기

- 프로세스는 사용자가 작업 관리자 등을 이용하여 임의로 죽일 수 있지만, 프로세스 안에서 동작하는 각 스레드는 프로세스와 같은 방법으로 죽일 수 없다.
  ➔살아 있는 스레드를 죽이려면 **Thread** 객체의 **Abort()** 메소드를 호출하면 된다.
  ➔**Abort()**를 사용하면 다른 스레드의 상황을 고려하지 않고 바로 **Exception**을 호출하여 스레드를 종료해버린다. 자원공유를 위해 **lock**가 걸린 상황이라면 아무도 접근 할 수 없는 상태가 되어 버리므로 문제가 될 수 있기 때문에 권장하지 않는다.

```
static void DoSomething()
{
    try
    {
        for(int i = 0; i < 10000; i++)
        {
            Console.WriteLine("DoSomething : {0}", i);
            Thread.Sleep(10);
        }
    }
    catch
    {
        //..
    }
    finally
    {
        //..
    }
```

```
static void Main(string[] args)
{
    Thread t1 =
        new Thread(new ThreadStart(DoSomething));

    t1.Start();
    t1.Abort();      // 스레드 취소(종료)
    t1.Join();
}
```

1. THREAD & TASK

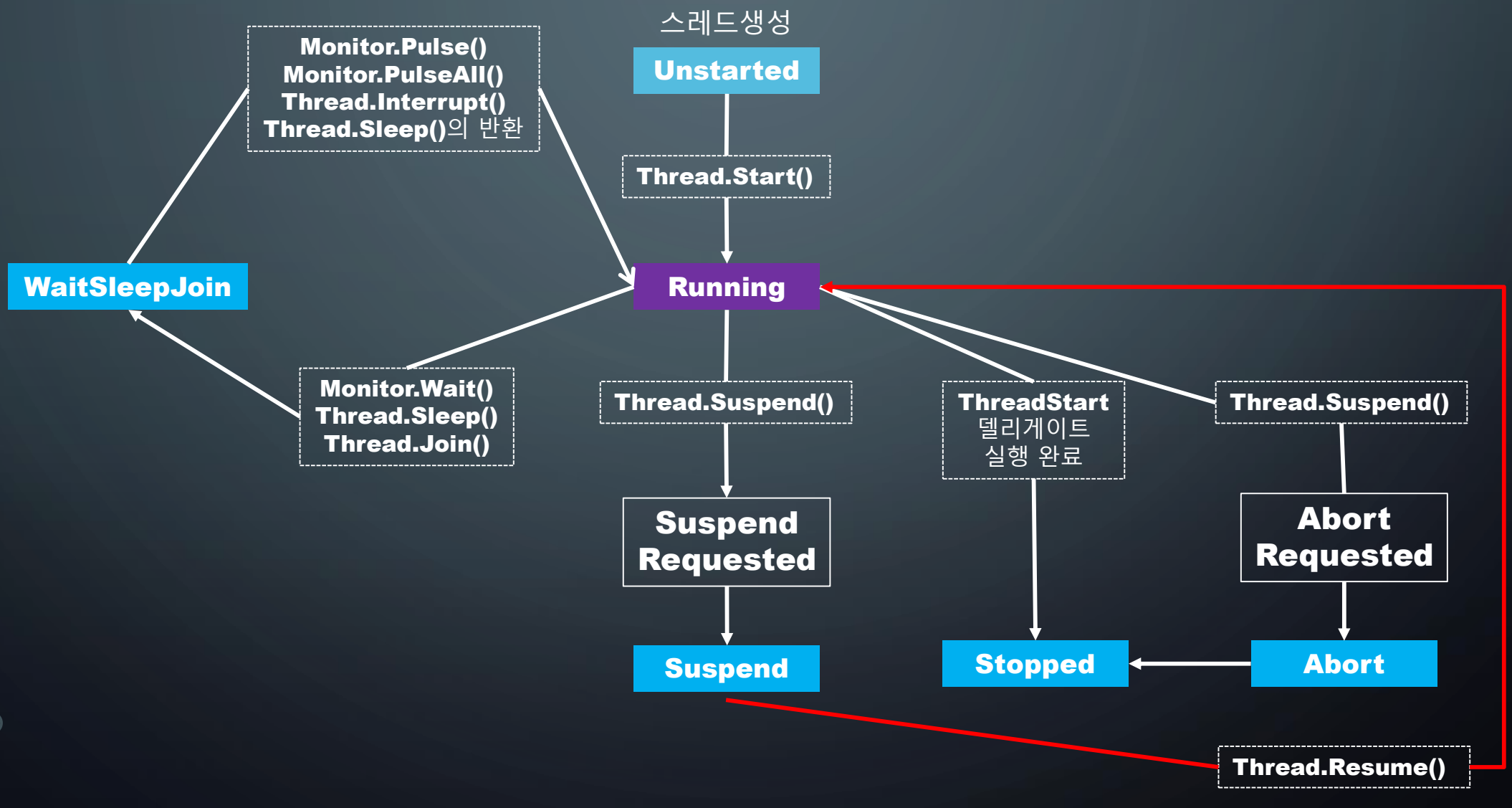
스레드의 상태 변화

- 스레드는 여러 상태 변화를 가지고 있다.
- **.NET Framework**는 스레드의 상태를 **ThreadState** 열거형에 정의해 두었다.

상태	설명
Unstarted	스레드 객체를 생성한 후 **Thread.Start()** 메소드가 호출되기 전의 상태
Running	스레드가 시작하여 동작 중인 상태를 나타낸다. **Unstarted** 상태의 스레드를 **Thread.Start()** 메소드를 통해 이상태로 만들 수 있다.
Suspended	스레드의 일시 중단 상태를 나타낸다. 스레드를 **Thread.Suspend()** 메소드를 통해 이 상태로 만들 수 있으며, **Suspended** 상태인 스레드는 **Thread.Resume()** 메소드를 통해 다시 **Running**상태로 만들 수 있다.
WaitSleepJoin	스레드가 블록된 상태를 나타낸다. **Block**이라는 이름이 아닌 이유는 **Wait, Sleep, Join**에 의해 블록 상태가 되기 때문이다.
Aborted	스레드가 취소된 상태를 나타낸다. **Thread.Abort()** 메소드를 호출하면 이 상태가 된다. **Aborted** 상태가 된 스레드는 다시 **Stopped** 상태로 전환되어 완전히 중지된다.
Stopped	중지된 스레드의 상태를 나타낸다. **Abort()** 메소드를 호출하거나 스레드가 실행중인 메소드가 종료되면 이 상태가 된다.
Background	스레드가 백그라운드로 동작하고 있음을 나타낸다. 포어그라운드**(Foreground)** 스레드는 하나라도 살아 있는 한 프로세스가 죽지 않지만, 백그라운드는 하나가 아니라 열 개가 살아 있어도 프로세스가 죽고 사는 것에는 영향을 미치지 않는다. 하지만 프로세스가 죽으면 백그라운드 스레드들도 모두 죽는다. **Thread.IsBackground** 속성에 **true** 값을 입력함으로써 스레드를 이 상태로 바꿀 수 있다.

1. THREAD & TASK

- 스레드의 상태는 서로 전환 될 수 있는 상태가 정해져 있다.
  ➔어떤 상태든 맘대로 변할 수 있는 것이 아니라 변화는 과정이 정해져 있다.



1. THREAD & TASK

ThreadState 열거형

- **Flags** 애트리뷰트를 갖고 있다.
  ➔**Flags**는 자신이 수식하는 열거형을 비트 필드**(Bit Field),** 즉 플래그 집합으로 처리할 수 있음을 나타낸다.

비트 필드(Bit Field)

- 한 바이트로 **0~255**까지 표현할 수 있는데 **0, 1, 2, 3, .., 7**정도의 값을 갖는 플래그를 표현하려고 한 바이트를 몽땅 사용할 필요가 없다.
  ➔비트 필들의 플래그로 비트연산에 의해 어떤 값이 포함되었는지 여부를 비트 논리 연산을 통해 알 수 있다.

```
enum MyEnum
{
        Apple,      //0
        Orange,     //1
        Kiwi,       //2
        Mango       //3
};

// 열거 요소에 대응하지 못하는 값은 형변환을
// 시도해도 원래 값으로 표현된다.
Console.WriteLine((MyEnum)0); // Apple
Console.WriteLine((MyEnum)1); // Orange
Console.WriteLine((MyEnum)2); // Kiwi
Console.WriteLine((MyEnum)3); // Mango
Console.WriteLine((MyEnum)4); // 4
Console.WriteLine((MyEnum)5); // 5
```

```
[Flags]
enum MyEnum
{
        Apple,      //0
        Orange,     //1
        Kiwi,       //2
        Mango       //3
};

// Flag 애트리뷰트는 열거형의 요소들의 집합으로
// 구성되는 값들도 표현할 수 있다.
Console.WriteLine((MyEnum)0); // Apple
Console.WriteLine((MyEnum)1); // Orange
Console.WriteLine((MyEnum)2); // Kiwi
Console.WriteLine((MyEnum)3); // Mango
Console.WriteLine((MyEnum)4); // Orange, Mango
Console.WriteLine((MyEnum)5); // Kiwi, Mango
```

1. THREAD & TASK

ThreadState의 상태표(비트 필드)

상태	10진수	2진수	상태	10진수	2진수
Running	0	000000000	Stopped	16	000010000
StopREquested	1	000000001	WaitSleepJoin	32	000100000
SuspendRequested	2	000000010	Suspended	64	001000000
Background	4	000000100	AbortRequested	128	010000000
Unstarted	8	000001000	Aborted	256	100000000

ThreadState 필드의 값을 확인하는 예.

```
if(t1.ThreadState & ThreadStaet.Aborted == ThreadState.Aborted)
    Console.WriteLine("스레드가 정지했습니다.");
else if(t1.ThreadState & ThreadState.Stopped == ThreadState.Stopped)
    Console.WriteLine("스레드가 취소되었습니다.");
```

1. THREAD & TASK

스레드 임의 종료 – 인터럽트(Thread.Interrupt)

스레드가 한참 동작 중인 상태(**Running**)를 피해서 **WaitJoinSleep** 상태에 들어갔을 때,
ThreadInterrupted Exception 예외를 던져 스레드를 중지시킨다.

```
static void DoSomething()
{
    try
    {
        for(int I = 0; I < 10000; i++)
        {
            Console.WriteLine("DoSomething : {0}", i);
            Thread.Sleep(10);
        }
    }
    catch(ThreadInterruptedException e)
    {
        //..
    }
    finally
    {
        //..
    }
}
```

```
static void Main(string[] args)
{
    Thread t1
        = new Thread(new ThreadStart(DoSomething));

    t1.Start();
    t1.Interrupt(); // 스레드 중단(종료)
    t1.Join();
}
```

1. THREAD & TASK

스레드 간의 동기화

- 각 스레드들은 다른 스레드들의 상황에는 관심이 없다. 자신이 항상 자원을 독점해야 하므로 서로 가져가려고 애쓴다. 그렇기 때문에 자원이 처리되어 갱신되기 전에 다른 곳에서 빼앗아 가버려서 원하는 자원에 대한 값 처리가 원활하게 이루어지지 않게 된다. 이런 상황을 개선하기 위해서 스레드 간의 동기화 방법을 사용하게 된다.
- **.NET Framework**에서는 **lock**키워드와 **Monitor** 클래스가 이 역할을 한다.
- 크리티컬 섹션을 지정해서 원하는 블록을 단 하나의 스레드만 접근하도록 허용하고 나머지는 기다리게 만들고 처리가 끝난 뒤 다른 스레드들이 접근 권한을 가지는 것이다.

lock 키워드

```
class Counter
{
        public int count = 0;
        public void Increase()
        {
                count = count + 1;
        }
}

Counter obj = new Counter();
Thread t1 = new Thread(new ThreadStart(obj.Increase));
Thread t2 = new Thread(new ThreadStart(obj.Increase));

T1.Start();
T2.Start();
T1.Join();
T2.Join();
```

```
class Counter
{
        public int count = 0;
        private readonly object thisLock = new object();
        public void Increase()
        {
                lock(thisLock)
                {
                        count = count + 1;
                }
        }
}
```

lock 키워드는 중괄호로 둘러싼 이 부분은 크리티컬 세션이 된다. 한 스레드가 이 코드를 실행하다가 **lock** 블록이 끝나는 괄호를 만나기 전까지는 다른 스레드는 절대 이 코드를 실행할 수 없다.

1. THREAD & TASK

Lock 키워드의 주의사항!

- **Lock** 키워드의 매개 변수로 사용하는 객체는 참조형이면 어느 것이든 쓸 수 있지만**, public** 키워드 등을 통해 외부 코드에서도 접근할 수 있는 경우에는 절대 사용하면 안된다**.**
  ➔ 문법적으로는 문제가 없으나 다른 자원에 대해 동기화를 해야 하는 스레드도 대기해버리는 상황이 된다**.**
- **3**가지 주의해야 하는 경우
  1. **this :** 클래스의 인스턴스는 클래스 내부 뿐만 아니라 외부에서도 자주 사용된다**.**
     ➔ **Lock(this)**는 금물**!**
  2. **Type** 형식 **: typeof**연산자나 **object** 클래스로부터 물려받은 **GetType()** 메소드는 **Type** 형식의 인스턴스를 반환한다**.** 즉**,** 코드의 어느 곳에서나 특정 형식에 대한 **Type** 객체를 얻을 수 있다**.**
     ➔ **typeof(SomeClass)**나 **lock(obj.GetType())**은 금물**!**
  3. **string** 형식 **:** 절대 **string** 객체로 **lock**를 하지 말자**. "abc"**는 어떤 코드에서든 얻어낼 수 있는 **string** 객체이기 때문이다**.**
     ➔ **lock("abc")** 금물**!**

1. THREAD & TASK

Monitor 클래스 동기화

Monitor.Enter()와 **Monitor.Exit()**를 이용해서 크리티컬 섹션을 설정 할 수 있다.

lock

```
public void Increase()
{
        int loopCount = 1000;
        while(loopCount-- > 0)
        {
                look(thisLock)
                {
                        count++;
                }
        }
}
```

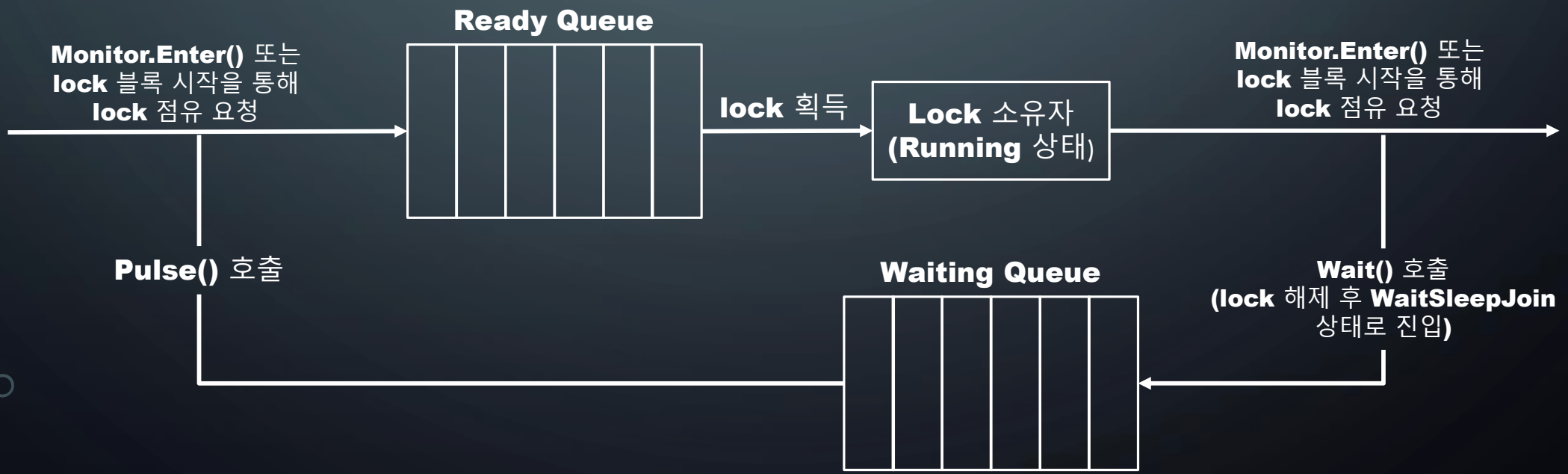
Monitor.Enter()와 Monitor.Exit()

```
public void Increase()
{
        int loopCount = 1000;
        while(loopCount-- > 0)
        {
                Monitor.Enter(thisLock);
                try
                {
                        count++;
                }
                finally
                {
                        Monitor.Exit(thisLock);
                }
        }
}
```

1. THREAD & TASK

Monitor.Wait()와 Monitor.Pulse() - 저수준 동기화

- **Lock**키워드만 사용할 때의 동기화 보다 더 섬세한 컨트롤이 가능하다.
  ➔ 그 만큼 구현도 까다롭다.
- 반드시 **lock** 블록안에서 호출해야 한다.
  ➔**lock**블록이 아닌 곳에서 호출하면 **CLR**이 **SynchronizationLockException**예외를 던진다.
- **Wait()** 메소드는 스레드를 **WaitSleepJoin** 상태로 만든다. 이렇게 **WaitSleepJoin** 상태에 들어간 스레드는 동기화를 위해 갖고 있던 **lock**을 내려놓은 뒤 **Waiting Queue**에 입력되고, 다른 스레드가 **lock**을 얻어 작업을 수행한다.
- 작업을 수행하던 스레드가 일을 마친 뒤 **Pulse()** 메소드를 호출하면 **CLR**은 **Waiting Queue**의 가장 첫 요소 스레드를 꺼낸 뒤 **Ready Queue**에 입력시킨다. **Ready Queue**에 입력된 스레드는 입력된 차례에 따라 **lock**을 얻어 **Running** 상태에 들어간다.



1. THREAD & TASK

Monitor.Wait()와 Monitor.Pulse()를 사용하는 패턴

Step1 클래스 안에 다음과 같은 동기화 객체 필드를 선언한다.

```
readonly object thisLock = new object();
```

Step2 스레드를 **WaitSleepJoin** 상태로 바꿔 블록시킬 조건(즉, **Wait()**를 호출한 조건)을 결정할 필드를 선언한다.

```
bool lockedCount = false;
```

Setp3 스레드를 블록 시키고 싶은 곳에서 **lock** 블록안에 선언한 **lockedCount**를 검사해서 **Monitor.Wait()**를 호출한다.

```
Lock(thisLock)
{
        while(lockedCount == true)
                Monitor.Wait(thisLock);
}
```

Step4 lockedCount의 조건으로 조정해서 **Count** 값을 변경하는 블록을 생성한다.

```
lockedCount = true;
count++;
lockedCount = false;

Monitor.Pulse(thisLock);
```

1. THREAD & TASK

Task와 Task<TResult>, Parallel

- 고성능 소프트웨어를 만들기 위해서는 멀티 코어를 활용해야 하며, 여러 개의 코어가 동시에 작업을 수행할 수 있도록 하는 병렬 처리 기법과 비동기 처리 기법이 있다.
- **.NET Framework**에서는 **System.Threading.Tasks** 네임페이스를 사용해서 병행 처리나 비동기 처리를 작성할 수 있도록 지원한다.
- **Thread(**여러 개의 작업을 각각 처리**)**와 달리 하나의 작업을 쪼개서 처리하는 작업에 유리하다.

병렬처리와 비동기 처리

- 하나의 작업을 여러 작업자가 나눠서 수행한 뒤 다시 하나의 결과로 만드는 것을 **병렬처리**라고 하고 작업 **A**를 시작한 후 **A**의 결과가 나올 때까지 마냥 대기하는 대신 곧이어 다른 작업 **B, C, D…**를 수행하다가 **A**가 끝나면 그때 결과를 받아내는 방법을 **비동기 처리**하고 한다.

System.Threading.Tasks.Task 클래스

- 명령을 내려서 작업을 수행하게 한 후에 원래 코드라인은 이 함수의 동작 여부에 관여하지 않고 별개로 코드 진행이 흘러간다.
- **Shoot And Forget**이라고 부르기도 한다.
- **async** 한정자와 **await** 연산자를 이용해 구현할 수도 있다.

1. THREAD & TASK

Action 델리게이트를 이용한 **Task** 예.

```
Action someAction = () =>
{
        Thread.Sleep(1000);
        Console.WriteLine("Printed asynchronously.");
};

// 생성자에서 넘겨받은 무명 함수를 비동기로 호출한다.
Task myTask = new Task(someAction);
myTask.Start();

Console.WriteLine("Printed synchronously.");

// myTask 비동기 호출이 완료될 때까지 기다린다.
myTask.Wait();

/* 결과는
Printed synchrously.
Printed asynchronously. */
```

1. THREAD & TASK

Task.Run() 메소드를 사용하는 예.

```
var myTask = Task.Run( () =>
{
      // Task의 생성과 시작을 단번에 한다.
      // Task가 실행할 Action 델리게이트도 무명 함수로 바꿨다.다.
      Tread.Sleep(1000);
      Console.WriteLine("Printed asynchronously.");
});

Console.WriteLine("Printed synchronously");

myTask.Wait();

/* 결과는
Printed synchronously.
Printed asynchronously. */
```

Task<TResult> 클래스

코드의 비동기 실행 결과를 손쉽게 얻을 수 있다.
Task와 달리 반환 받을 수 있는 형태인 **Func** 델리게이트 형식으로 주로 사용한다.

1. THREAD & TASK

Func 델리게이트 사용 예.

```
var myTask = Task<List<int>>.Run( () =>
{
        Thread.Sleep(1000);

        List<int> list = new List<int>();
        list.Add(3);
        list.Add(4);
        list.Add(5);

        // Task<Tresult>는 Tresult 형식의 결과를 반환한다.
        return list;
});

var myList = new List<int>();

myList.Add(0);
myList.Add(1);
myList.Add(2);

// myList의 요소는 0, 1, 2, 3, 4, 5가 된다.
myTask.Wait();
myList.AddRange(myTask.Result.ToArray());
```

1. THREAD & TASK

Parallel 클래스

- **System.Threading.Tasks** 네임스페이스의 클래스
- **For(), Foreach()**등의 메소드를 제공해서 병렬처리를 더 쉽게 구현하게 해 준다.

사용 예.

```
Void SomeMethod(int i)
{
      Console.WriteLine(i);
}

// ..
Parallel.For(0, 100, SomeMethod);
```

async 한정자와 await 연산자를 이용한 비동기 코드

- **async** 한정자는 메소드, 이벤트 처리기, 태스크, 람다식 등을 수식함으로써 **C#** 컴파일러가 이들을 호출하는 코드를 만날 때 호출 결과를 기다리지 않고 바로 다음 코드로 이동하도록 실행 코드를 생성하게 한다.

async 한정자 사용 예.

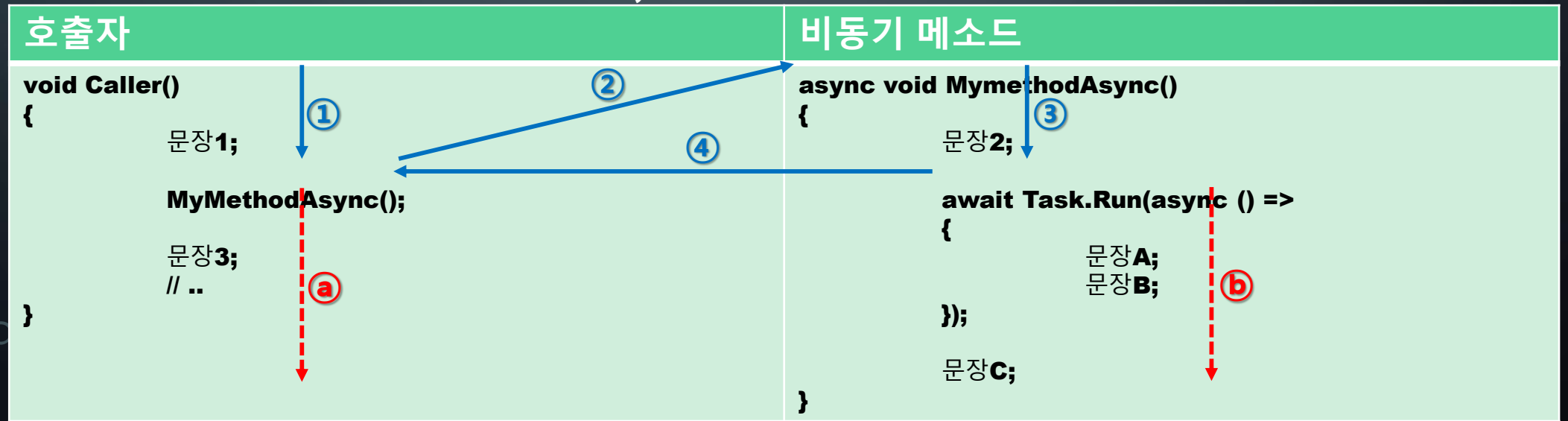
```
Public static async Task MyMethod()
{     //...         }
```

1. THREAD & TASK

- **async**로 선언한 **void** 형식의 메소드는 호출 즉시 호출자에게 제어를 돌려준다. **Void** 메소드는 **async** 한정자 하나만으로도 완전한 비동기 코드가 되는 것이다. 하지만 **Task.Task<TResult>** 형식의 메소드는 **async**로 수식하기만 해서는 보통의 동기 코드와 다름없이 동작한다.
- **C#** 컴파일러는 **Task** 또는 **Task<TResult>** 형식의 메소드를 **async** 한정자가 수식하는 경우, **await** 연산자가 해당 메소드 내부의 어디에 위치하는지를 찾는다.
- **await** 연산자를 찾으면 그곳에서 호출자에게 제어를 돌려주도록 실행 파일을 만든다.
  ➔ **await** 연산자를 만나지 못한다면 호출자에게 제어를 돌려주지 않으므로 그 메소드/태스크는 동기적으로 실행하게 된다.

결론,

async로 한정한 **void**형식 메소드는 **await** 연산자가 없어도 비동기로 실행된다.
async로 한정한 **Task** 또는 **Task<TResult>**를 반환하는 메소드/태스크/람다식은 **await** 연산자를 만나는 곳에서 호출자에게 제어를 돌려주며, **await** 연산자가 없는 경우 동기로 실행 된다.



1. THREAD & TASK

앞의 구문에서 **Caller()**의 실행이 시작되면, ①의 흐름을 따라 문장**1**이 실행되고, 이어서 ②를 따라 **MyMethodAsync()** 메소드의 실행으로 제어가 이동한다. **MyMethodAsync()** 에서는 ③을 따라 문장**2**가 실행되고 나면 **async** 람다문을 피연산자로 하는 **await** 연산자를 만나게 된다. 여기서 **CLR**은 ④를 따라 제어를 호출자인 **Caller()**에게로 이동시키고, 앞의 구문에서 점선으로 표시되어 있는 ⓐ와 ⓑ의 흐름을 동시에 실행하게 된다.

.NET Frame워크가 제공하는 비동기 API

많은 **API** 메소드가 존재 한다.
System.IO.Stream 클래스가 제공하는 **~Async()** 형태의 메소드가 제공된다면 비동기 메소드가 존재하는 것들이다.

Ready / Writed의 동기 비동기 버전 메소드

동기 버전 메소드	비동기 버전 메소드	설명
Read	**ReadAsync**	스트림에서 데이터를 읽는다
Write	**WriteAsync**	스트림에 데이터를 기록한다.

1. THREAD & TASK

동기화 / 비동기화 코드 예.

동기버전

```
static long CopySync(string FromPath, string ToPath)
{
        using( var fromStream = new FileStream(FromPath, FileMode.Open);
        {
                long totalCopied = 0;
                using(var toStream = new FileStream(ToPath, FileMode.Create))
                {
                        byte[] buffer = new byte[1024];
                        int nRead = 0;
                        while((nRead = fromStream.Read(buffer, 0, buffer.Length)) != 0)
                        {
                                toStream.Write(buffer, 0, nRead);

                                //Read() 메소드와 Write() 메소드
                                totalCopied += nRead;
                        }
                }
                return totalCopied
        }
}
```

1. THREAD & TASK

동기화 / 비동기화 코드 예.

비동기버전

```
// async로 한정한 코드를 호출하는 코드도 역시 async로 한정되어 있어야한다.
// 반환형식은 Task 또는 void 형이어야 한다.
async Task<long> CopyAsync(string FromPath, string ToPath)
{
        using(var fromStream = new FileStream(FromPath, FileMode.Open))
        {
                long totalCopied = 0;
                using(var toStream = new FileStream(ToPath, FileMode.Create))
                {
                        byte[] buffer = new byte[1024];
                        int nRead = 0;

                        // ReadAsync()와 WriteAsync() 메소드는 .NET Framework에 async로
                        // 한정되어 있다. 이들을 호출하려면 await 연산자가 필요하다.
                        while ((nRead = await fromStream.ReadAsync
                                                        (buffer, 0, buffer.Length)) != 0)
                        {
                                await toStream.WriteAsync(buffer, 0, nRead);
                                totalCopied += nRead;
                        }
                }
                return totalCopied;
        }
}
```