

# 이번 강의에서 다루는 내용

- 프로세스와 스레드
- 멀티 스레딩
- 스레드의 시작과 종료
- 스레드의 상태 변화
- 스레드간의 동기화
- Task 클래스
- Parallel 클래스
- async와 await

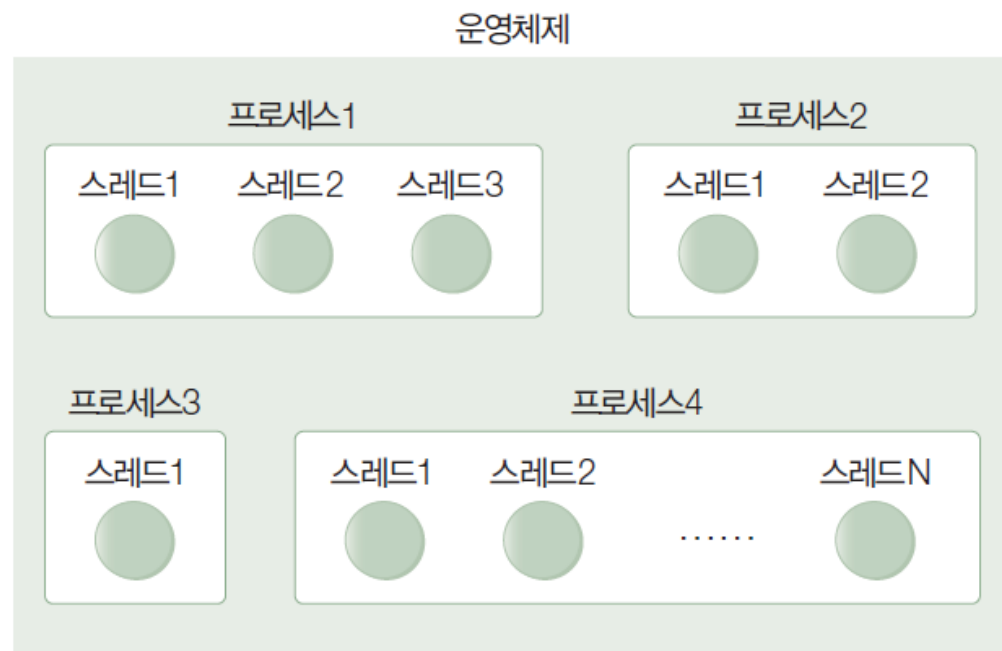
# 프로세스와 스레드

- **프로세스(Process)**

- 실행파일의 데이터와 코드가 메모리에 적재되어 동작하는 것
- word.exe가 실행 파일이라면, 이 실행 파일을 실행한 것이 프로세스

- **스레드(Thread)**

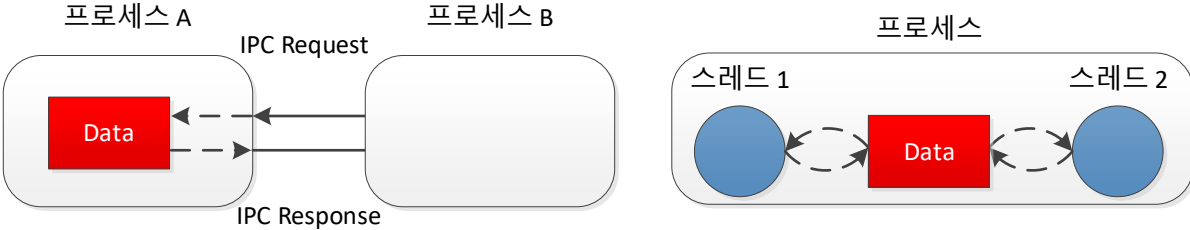
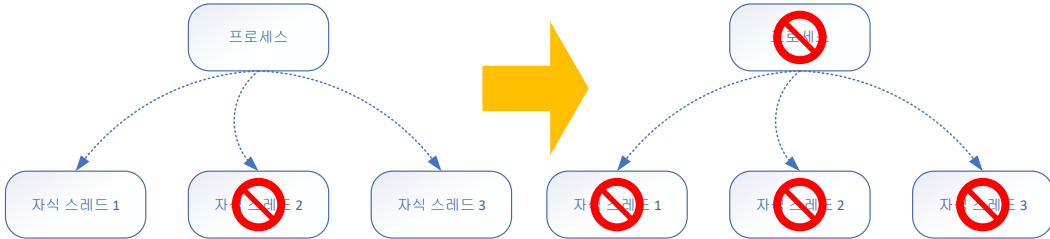
- 스레드는 운영체제가 CPU 시간을 할당하는 기본 단위
- 프로세스가 뱃줄이라면 스레드는 뱃줄을 이루는 실



[운영체제와 프로세스, 프로세스와 스레드]

# 멀티 스레드

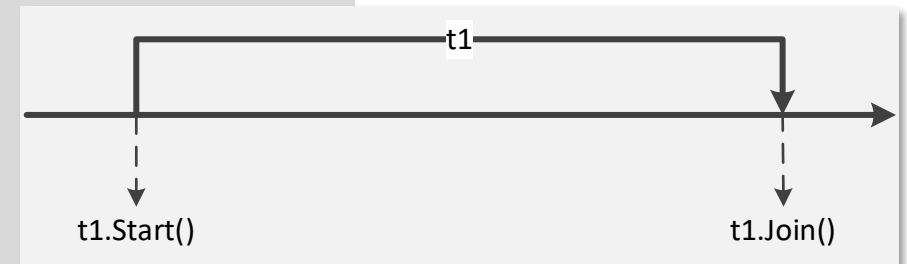
- 한 프로세스 안에 **2개 이상의 스레드를 실행**하는 기법

|    |   |  |
|----|---|--|
| 장점 | <ol style="list-style-type: none"> <li>1. 높은 경제성</li> <li>2. 높은 사용자 응답성</li> <li>3. 용이한 데이터 교환</li> </ol>               |  <p>The diagram illustrates two scenarios of multi-threading. On the left, '프로세스 A' (Process A) and '프로세스 B' (Process B) are shown. Process A contains a red box labeled 'Data'. They are connected by two arrows: a solid arrow labeled 'IPC Request' pointing from A to B, and a dashed arrow labeled 'IPC Response' pointing from B to A. On the right, a single '프로세스' (Process) is shown containing two blue circles labeled '스레드 1' (Thread 1) and '스레드 2' (Thread 2). Both threads are connected to a central red box labeled 'Data' with dashed double-headed arrows, indicating shared data within the process.</p>  |
| 단점 | <ol style="list-style-type: none"> <li>1. 높은 개발 난이도</li> <li>2. 과다한 스레드는 성능 저하</li> <li>3. 스레드의 문제는 프로세스로 확산</li> </ol> |  <p>The diagram shows a flow of a problem from one process to others. On the left, a '프로세스' (Process) box has three dashed arrows pointing to '자식 스레드 1' (Child Thread 1), '자식 스레드 2' (Child Thread 2), and '자식 스레드 3' (Child Thread 3). '자식 스레드 2' has a red circle with a diagonal line through it, indicating a problem. A large yellow arrow points to the right, where a similar structure is shown. The top '프로세스' box now also has a red circle with a diagonal line through it. The three child threads below it ('자식 스레드 1', '자식 스레드 2', '자식 스레드 3') all have the same red circle with a diagonal line through it, showing that the problem has spread to the entire process.</p> |

# 스레드 시작하기

- **System.Threading.Thread** : 스레드를 나타내는 클래스
- 스레드 기동 절차
  - ① Thread 인스턴스 생성(스레드가 실행할 메소드를 생성자 인수로 입력)
  - ② Thread.Start() 메소드 호출(스레드 시작)
  - ③ Thread.Join() 메소드 호출(스레드 종료 대기)

```
static void DoSomething()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine("DoSomething : {0}", i);
    }
}
static void Main(string[] args)
{
    Thread t1 = new Thread(new ThreadStart(DoSomething));
    t1.Start();
    t1.Join();
}
```



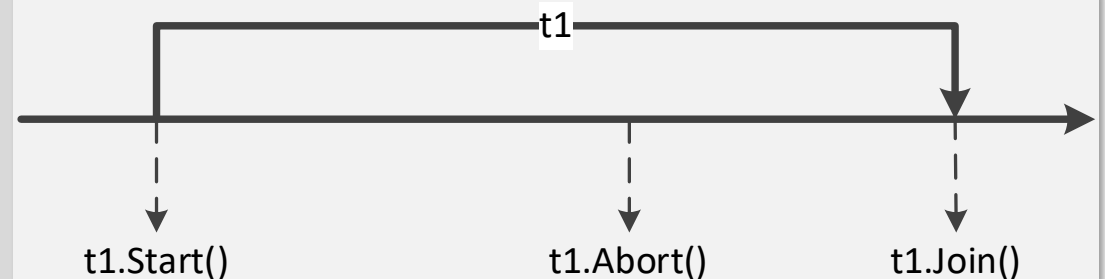
# 스레드 멈추기-Abort

- 스레드는 실행중인 메소드가 종료되면 함께 종료되지만, 필요한 경우 **Abort()** 메소드를 호출하여 강제 종료 가능
- **Abort()** 메소드 호출시 **ThreadAbortedException** 발생

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

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

    t1.Start();
    t1.Abort();
    t1.Join();
}
```



# 스레드 멈추기-Interrupt

- `Thread.Interrupt()` 메소드는 스레드가 `WaitJoinSleep` 상태에 진입했을 때 `ThreadInterruptedException` 예외를 일으켜 스레드를 중단시킴
- 스레드가 블록되어 있을 때 스레드를 중단시키므로 부작용을 최소화할 수 있음

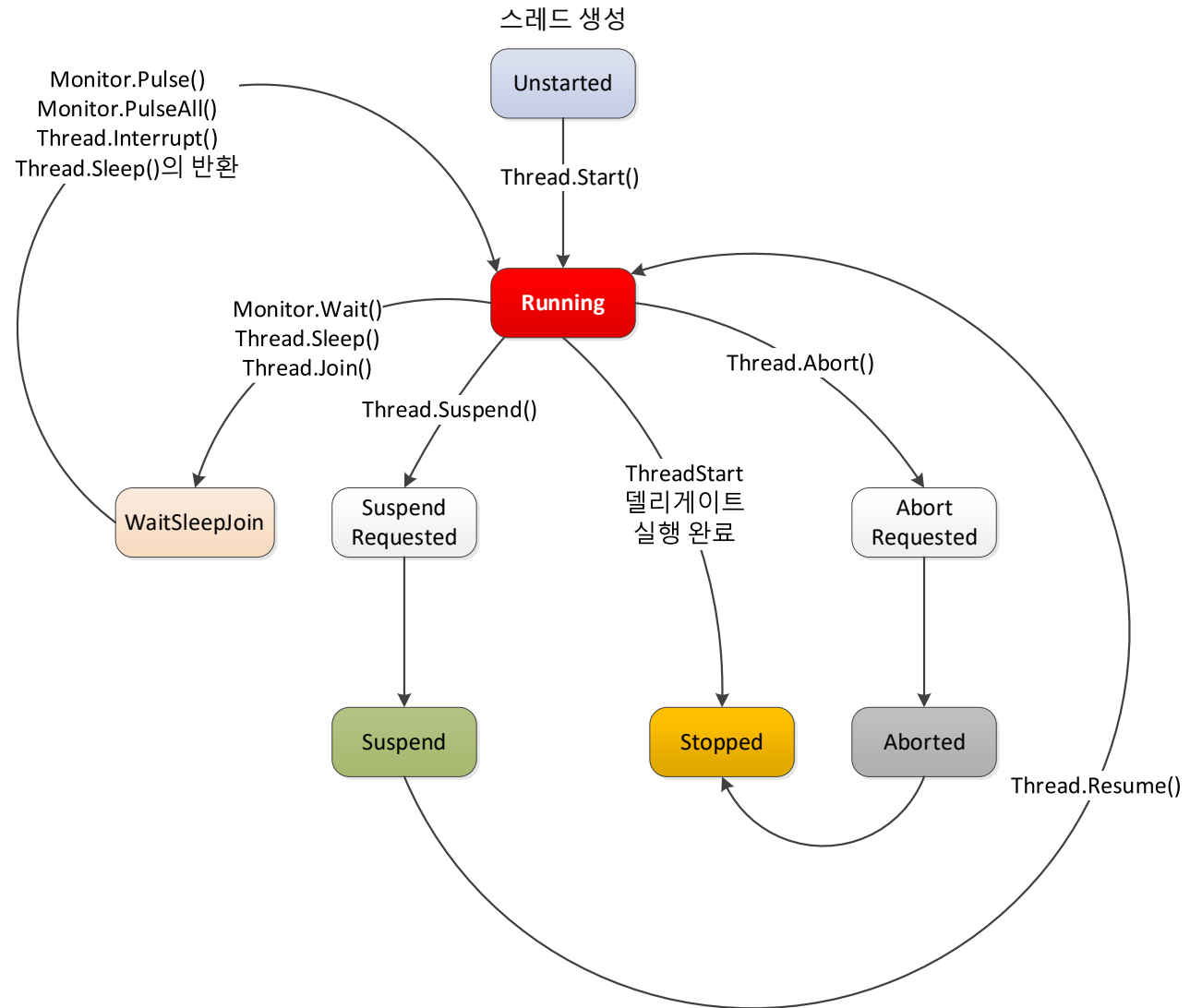
```
static void DoSomething()
{
    try
    {
        for (int i = 0; i < 10000; i++)
        {
            Console.WriteLine("DoSomething : {0}", i);
            Thread.Sleep(10);
        }
    }
    catch (ThreadInterruptedException e)
    { /*...*/ }
    finally
    { /*...*/ }
}
```

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

    t1.Start();
    t1.Interrupt();
    t1.Join();
}
```

# 스레드의 상태 변화

- **Unstarted**: 스레드 생성 직후
- **Running**: 실행중
- **WaitSleepJoin**: 블록된 상태
- **Suspend**: 일시 중단 상태
- **Aborted**: 취소 상태
- **Stopped**: 정지 상태



# 스레드간의 동기화

- 멀티스레드 동기화(Synchronization)

- 자원(예:변수)을 한 번에 한 스레드가 사용하도록 순서를 맞추는 것

- 동기화가 없다면?

- 단일 연산이라고 믿었던 C# 코드도 IL로 변환해보면 복잡한 단계를 거침
- 스레드 A가 i++을 실행하는 중에 스레드 B가 끼어들어 i++을 먼저 실행한다면?

|                              |  |
|------------------------------|--|
| <pre>int i=0;<br/>i++;</pre> | <pre>IL_0001: ldc.i4.0<br/>IL_0002: stloc.0<br/>IL_0003: ldloc.0<br/>IL_0004: ldc.i4.1<br/>IL_0005: add<br/>IL_0006: stloc.0</pre> |
|------------------------------|--|

- .NET이 제공하는 대표적 동기화 도구

- lock 키워드
- Monitor 클래스



# 크리티컬 섹션 (Critical Section)

- 동시 접근이 불가능하도록 보호된 코드 영역
- C#에서는 `lock` 키워드를 이용하여 생성 가능

```
class Counter
{
    public int count = 0;
    private readonly object thisLock = new object();

    public void Increase()
    {
        lock ( thisLock )
        {
            count = count + 1;
        }
    }
}

MyClass obj = new MyClass();
Thread t1 = new Thread(new ThreadStart(obj.Increase));
Thread t2 = new Thread(new ThreadStart(obj.Increase));
Thread t3 = new Thread(new ThreadStart(obj.Increase));

t1.Start(); t2.Start(); t3.Start();
t1.Join(); t2.Join(); t3.Join();

Console.WriteLine( obj.count );
```

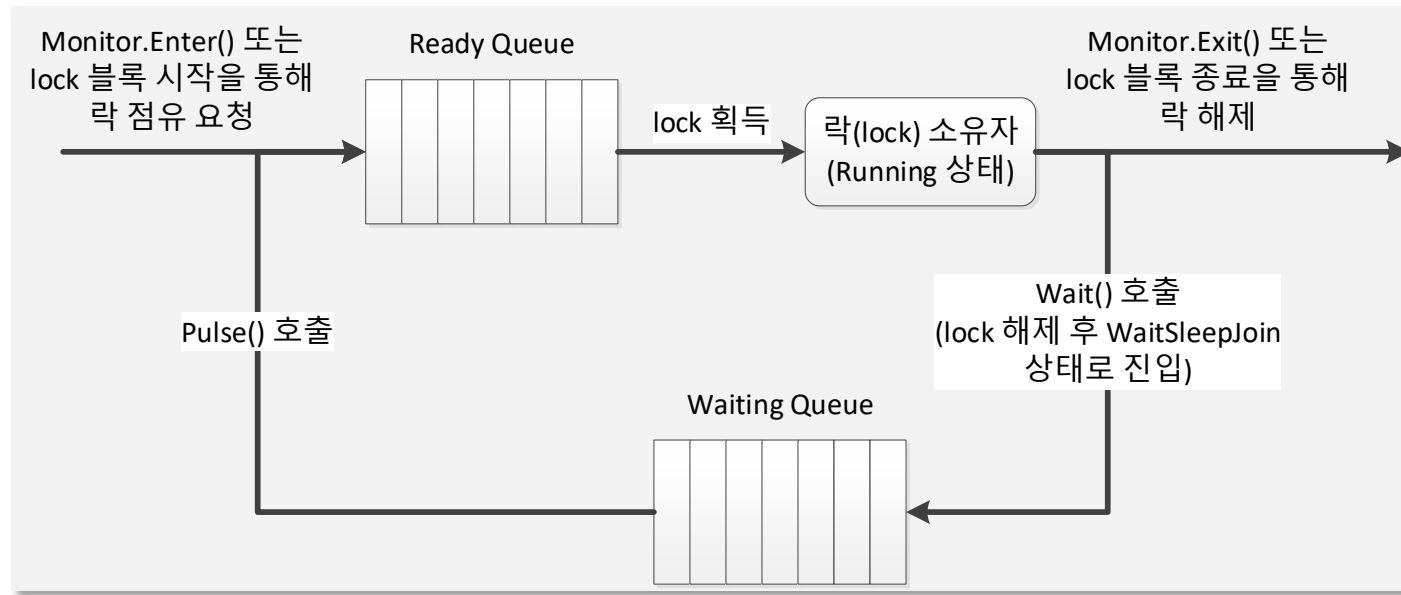
# Monitor 클래스

- 스레드 동기화를 지원하는 메소드 제공
- Monitor.Enter()와 Monitor.Exit()를 이용하면 lock과 같이 크리티컬 섹션 생성 가능

| lock 이용   | Monitor 이용   |
|---|--|
| <pre>public void Increase() {     int loopCount = 1000;     while (loopCount-- &gt; 0)     {         lock (thisLock)         {             count++;         }     } }</pre> | <pre>public void Increase() {     int loopCount = 1000;     while (loopCount-- &gt; 0)     {         Monitor.Enter(thisLock);         try         {             count++;         }         finally         {             Monitor.Exit(thisLock);         }     } }</pre> |

# Monitor.Wait()와 Monitor.Pulse()

1. Monitor.Wait() 메소드 호출을 통해 스레드는 WaitSleepJoin 상태로 진입
2. WaitSleepJoin 상태의 스레드는 Waiting Queue에 입력
3. Running 상태 스레드가 작업을 마친 뒤 Monitor.Pulse() 메소드 호출
4. CLR은 Waiting Queue의 가장 첫 요소 스레드를 꺼내 Ready Queue에 입력
5. Ready Queue에 입력된 스레드는 lock을 획득하여 Running 상태에 진입



# System.Threading.Tasks.Task 클래스

- Action 대리자를 실행
- Start() 메소드 : Action 대리자 비동기 실행
- Factory.StartNew() 메소드 : Task 객체 생성 및 Action 대리자 비동기 실행
- Wait() 메소드 : Action 대리자 실행 완료 대기

```
var myTask = Task.Factory.StartNew(  
    ()=>  
    {  
        Thread.Sleep(1000);  
        Console.WriteLine(" Printed asynchronously.");  
    }  
);  
  
Console.WriteLine("Printed synchronously.");  
myTask.Wait();
```

Printed synchronously.  
Printed asynchronously,

# System.Threading.Tasks.Task<TResult> 클래스

- Func 대리자 실행
- Start() 메소드 : Func 대리자 비동기 실행
- Factory.StartNew() 메소드 : Task 객체 생성 및 Func 대리자 비동기 실행

```
var myTask = Task<List<int>>.Factory.StartNew(  
    () =>  
    {  
        Thread.Sleep(1000);  
        List<int> list = new List<int>();  
        list.Add(3);  
        list.Add(4);  
        list.Add(5);  
        return list;  
    }  
);
```

```
var myList = new List<int>();  
myList.Add(0);  
myList.Add(1);  
myList.Add(2);  
  
myTask.Wait();  
  
myList.AddRange(myTask.Result.ToArray());
```

# Parallel 클래스

- 병렬 반복 코드를 지원하는 클래스
- `Parallel.For()/Parallel.ForEach()` 메소드에 반복 범위와 `Action` 대리자를 인수로

```
bool IsPrime(long number) { /* ... */}

// ...
{
    int from = 0;
    int to = 10000000000;
    List<long> total = new List<long>();

    Parallel.For(from, to, (long i) =>
        {
            if (IsPrime(i))
                lock(total)
                    total.Add(i);
        });
}
```

# async와 await (1/2)

- async 한정자
  - async 한정자는 메소드, 이벤트 처리기, 태스크, 람다식 등을 수식함
  - C# 컴파일러가 async 한정자로 수식한 코드의 호출자를 만날 때 호출 결과를 기다리지 않고 바로 다음 코드로 이동하도록 실행 코드를 생성
- async로 한정하는 메소드는 반환 형식이 Task나 Task<TResult>, 또는 void여야 함.
  - 실행하고 잊어버릴(Shoot and Forget) 작업을 담고 있는 메소드라면 반환형식을 void로 선언
  - 작업이 완료될 때까지 기다리는 메소드라면 Task, Task<TResult>로 선언

# async와 await (2/2)

- async로 한정한 Task 또는 Task<TResult>를 반환하는 메소드/태스크/람다식은 await 연산자를 만나는 곳에서 호출자에게 제어 반환
- await 연산자가 없는 경우엔 동기로 실행

