HCI 프로그래밍

15. C# Thread

HCI
Human Computer Interaction

Multi-tasking vs Thr

멀티태스킹 **(Multi-tasking)**

- 하나의 응용프로그램이 여러 개의 작업 (태스크)을 동시에 처리

스레드 **(Thread)**

- 마치 바늘이 하나의 실 (thread)을 가지고 바느질하는 것과 스레드는 일맥 상통함

Process vs Thread

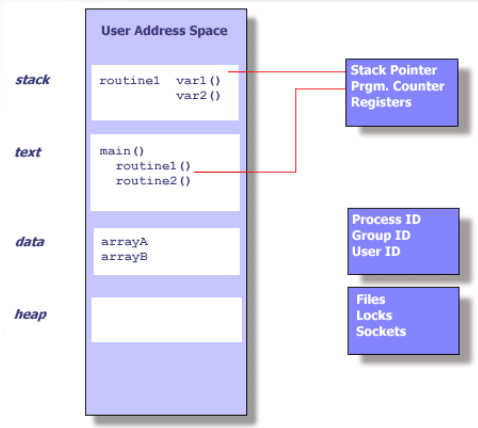
프로세스 (process)

- 운영체제로부터 process를 할당받고,
  운영되기 위해 필요한 주소공간, 메모리 등 자원을 할당받음

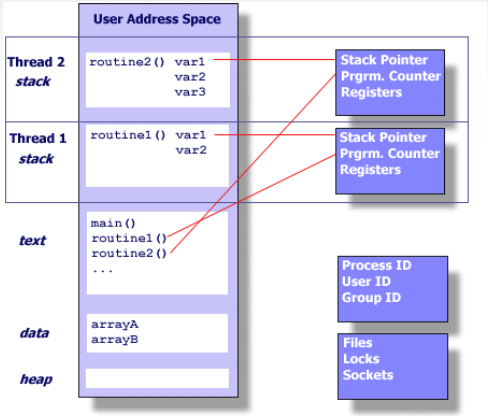
Process vs Thread

스레드 (thread)

- 한 프로세스 내에서 동작되는 여러 실행의 흐름으로, 같은 process 내의 주소공간이나 자원들을 thread끼리 공유하면서 실행됨



UNIX PROCESS

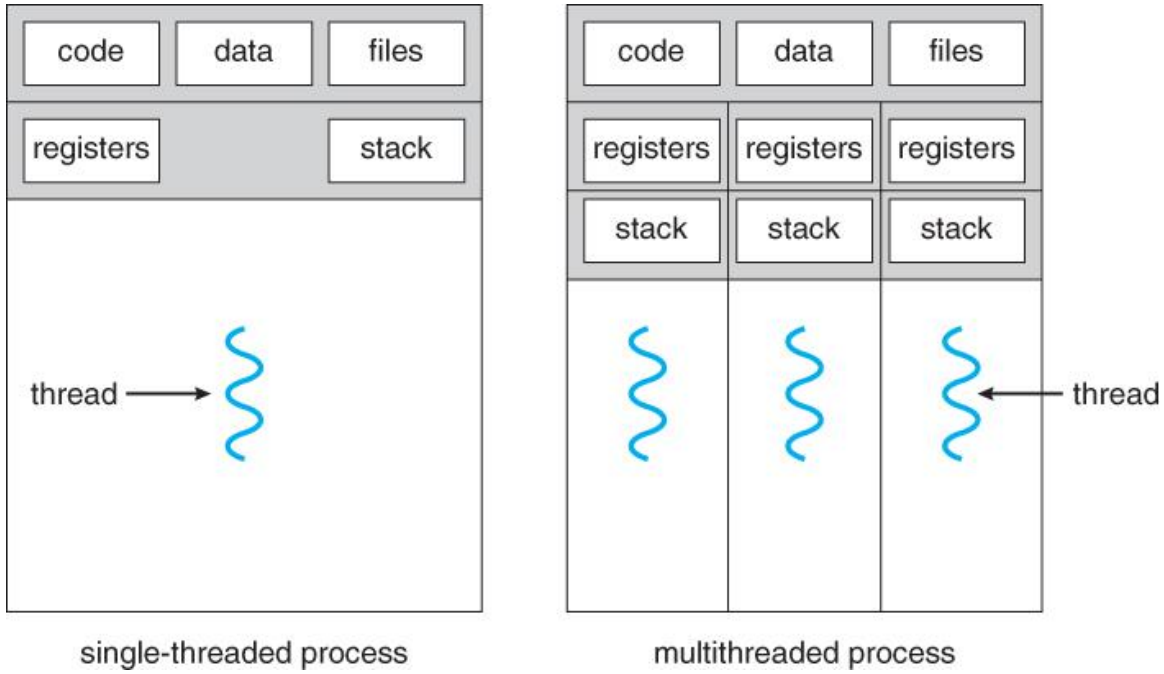


THREADS WITHIN A UNIX PROCESS

https://computing.llnl.gov/tutorials/pthreads/

Single vs Multithreaded Process

- **Threads are light-weight processes within a process**



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Why Thread?

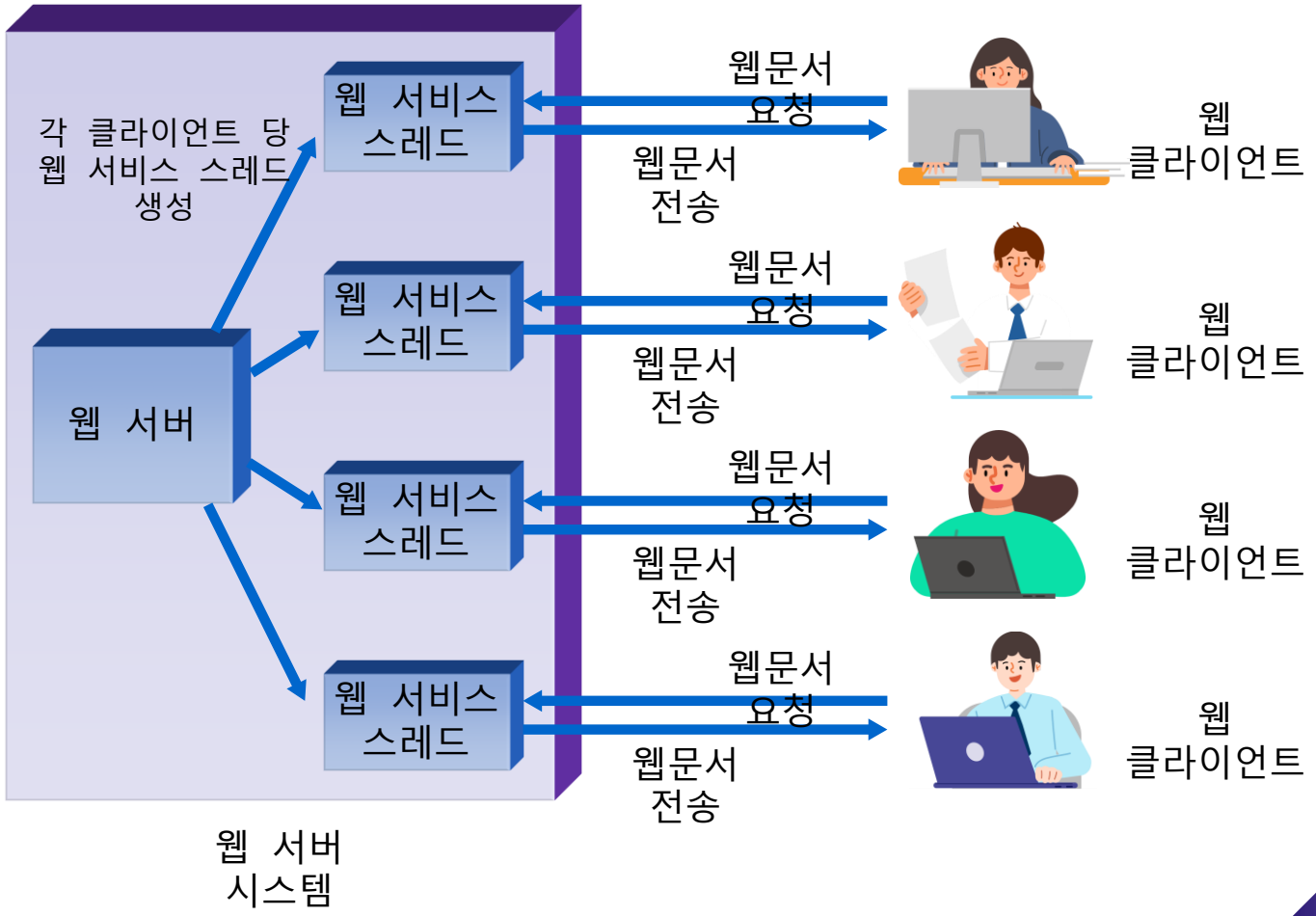
프로세스 (process)

- 프로세스를 생성하거나 파괴하는 데 비용이 많이 듦
- 더 많은 메모리 공간이 필요함
- 프로세스 복원 또는 교환을 위한 비용이 많이 듦
- 메모리 맵 변경이 필요함

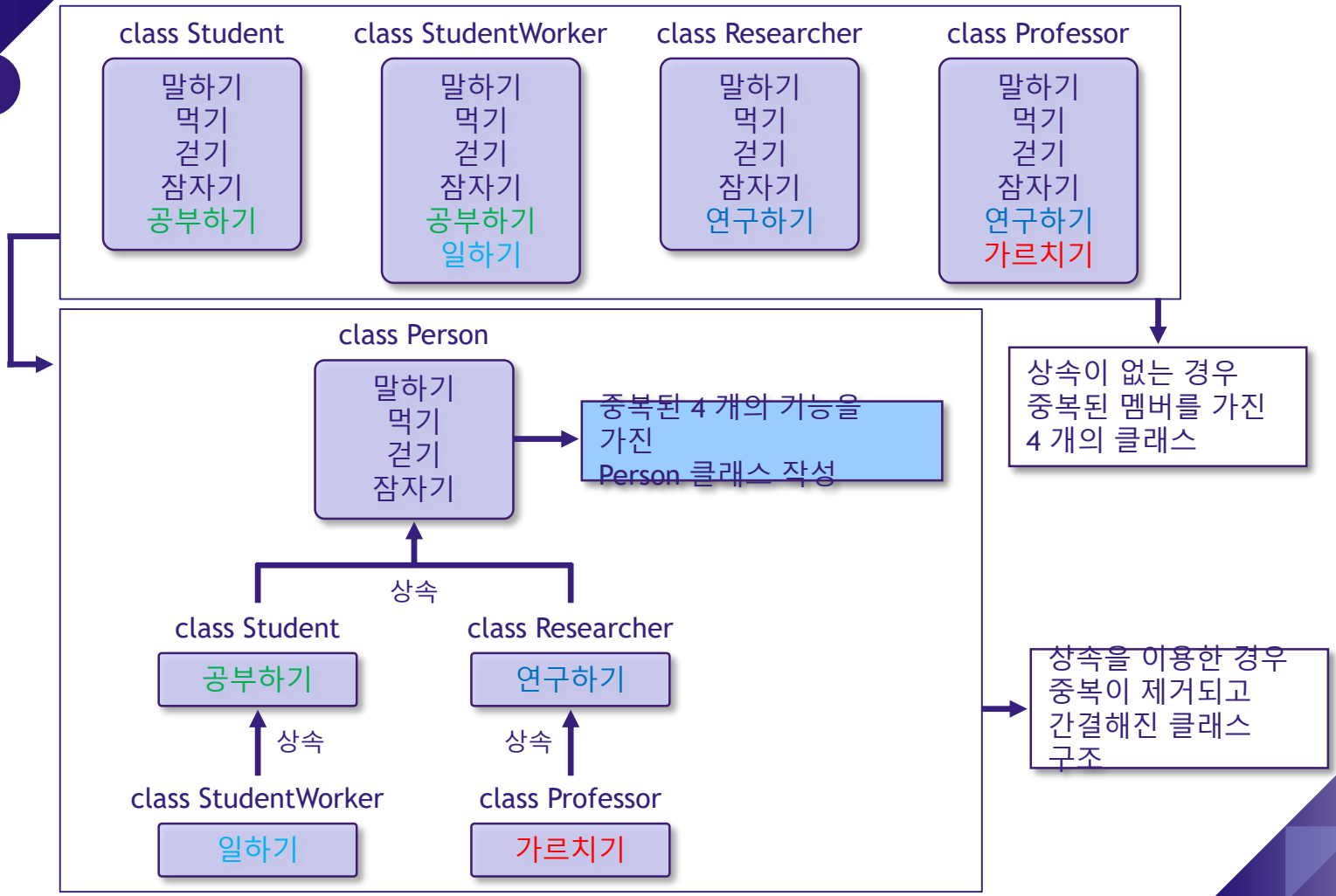
스레드 (Thread)

- 스레드 풀 (a pool of threads)을 유지하고 다시 사용 (reuse)할 수 있음
- 메모리 공간이 공유되며 항상 스왑 될 필요는 없음
- 프로세스의 모든 자원을 공유함

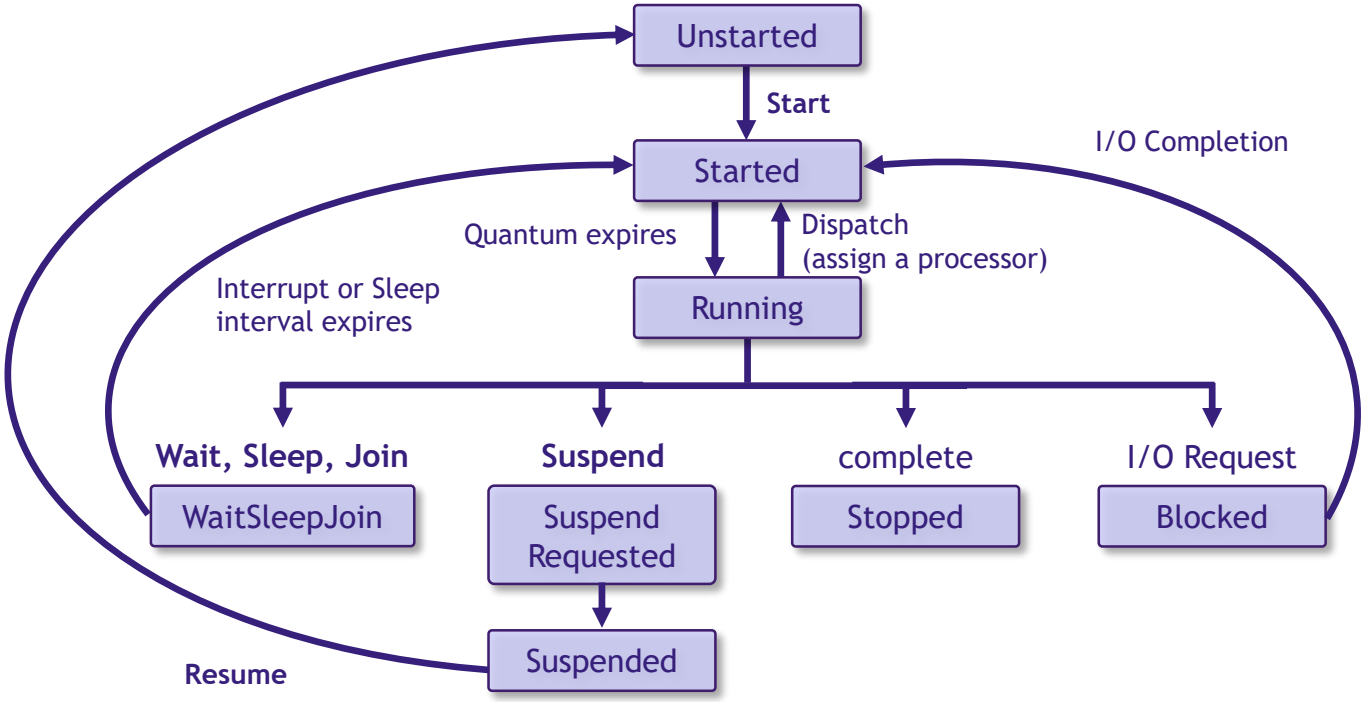
Threaded Application



Lifecycle of Threads in .NET



Lifecycle of Threads in .NET



Thread State

상태	설명
Running	스레드가 시작되었고 차단되지 않았으며 보류 중인 스레드가 없음 ThreadAbortException
StopRequested	스레드를 중지하도록 요청 내부전용
SuspendRequested	스레드를 일시 중단하도록 요청
Background	스레드는 전경 스레드가 아닌 배경 스레드로 동작됨 이 상태는 Thread.IsBackground 속성을 설정하여 제어함
Unstarted	스레드 객체를 생성한 후 Thread.Start 메소드 호출 전 상태
Stopped	스레드가 중지됨
WaitSleepJoin	스레드가 차단됨 Thread.Sleep 또는 Thread.Joint (Monitor.Enter 또는 Monitor.Wait) 하거나 ManualResetEvent
Suspended	스레드가 일시 중단됨
AbortRequested	Thread.Abort 메소드가 스레드에서 호출
Aborted	스레드가 취소됨 AbortRequested하고 스레드는 이제 죽었지만 상태는 아직 Stopped로 변경되기 이전

Thread 클래스

주요 멤버	설명
void Start()	해당 스레드의 실행
void Abort()	해당 스레드의 종료
bool Join (int millisecondsTimeout)	해당 스레드의 실행 종료 시까지 대기
void Suspend()	해당 스레드를 대기상태로 변경
void Resume()	해당 스레드를 실행상태로 변경
static void Sleep (int millisecondsTimeout)	해당 스레드를 특정 시간 동안 대기상태로 변경
CurrentThread	현재 실행중인 스레드 반환
IsAlive	해당 스레드의 실행여부 반환
IsBackground	해당 스레드가 Background인지 반환
Name	해당 스레드의 이름 설정 및 반환
Priority	스레드 우선순위
ThreadState	스레드 상태 반환

Thread Priority

우선순위	설명
Highest	가장 높은 우선권
AboveNormal	높은 우선권
Normal	평균 우선권
BelowNormal	낮은 우선권
Lowest	가장 낮은 우선권

MainThread

- **System.Threading.Thread** 클래스는
  멀티스레드 작업에 사용됨
- 프로세스에서 실행되는 첫 번째 스레드는 **MainThread**

```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class Program {
      static void Main(string[] args) {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";
         Console.WriteLine("This is {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

Thread 생성

스레드 생성 방법

- System.Threading.Thread 클래스
- public delegate void ThreadStart() 또는 public delegate void ParameterizedThreadStart(object obj) 사용

Thread 생성

생성자

- Thread(ParameterizedThreadStart)
  스레드가 시작될 때 스레드로 개체가 전달될 수 있도록 하는 대리자를 지정하여 스레드 클래스의 새 인스턴스를 초기화
- Thread(ThreadStart)
  ThreadStart 대리자를 지정하여 스레드 클래스의 새 인스턴스를 초기화
- Thread(ParameterizedThreadStart, Int32)
- Thread(ThreadStart, Int32)

스레드 시작

- Start()

Thread 생성

- 스레드는 **Thread** 클래스를 사용하여 생성 후, **Start()** 호출

```
using System;
using System.Threading;

namespace MultithreadingApplication {
  class Program {
    public static void CallToChildThread() {
      Console.WriteLine("Child thread starts");
    }
    static void Main(string[] args) {
      Console.WriteLine(“Main thread");
      ThreadStart ts = new ThreadStart(CallToChildThread);
      Thread t = new Thread(ts);
      t.Start();
      Console.ReadKey();
    }
  }
}
```

Thread 생성의 다양한 예제

```
public static void Run() {
    Console.WriteLine("Child thread run");
}
static void Main(string[] args) {
    // ThreadStart delegate 객체 생성 후 Thread 생성자에 전달
    Thread t1 = new Thread(new ThreadStart(Run));
    t1.Start();
    // 컴파일러가 Run() 메소드에서 ThreadStart delegate 추론하여 생성
    Thread t2 = new Thread(Run);
    t2.Start();
    // 익명 메소드를 사용하여 생성
    Thread t3 = new Thread(delegate() {
        Run();
    } );
    t3.Start();
    // 람다식을 사용하여 생성
    Thread t4 = new Thread(() => Run());
    t4.Start();
    // 람다식을 사용하여 생성
    new Thread(() => Run()).Start();
}
```

Thread 생성의 다양한 예제

```
class MyClass {
    public void Run() {
        Console.WriteLine("MyClass run");
    }
}
class Program {
    static void Main(string[] args) {
        // MyClass 클래스의 Run 메소드 호출
        MyClass mc = new MyClass();
        Thread t = new Thread(mc.Run);
        t.Start();
    }
}
```

Thread 클래스 인자 전달

- public delegate void ParameterizedThreadStart(object obj)는 하나의 object 인자를 전달하고 반환이 없는 형식

```
class Program {
    public static void Calc(object radius) {
        Console.WriteLine("r={0} area={1}", radius, radius * radius * 3.14);
    }
    public static void Sum(int i, int j, int k) {
        Console.WriteLine("sum={0}", (i + j + k));
    }
    static void Main(string[] args) {
        // ParameterizedThreadStart  인자 전달 & Start 인자 전달
        Thread t1 = new Thread(new ParameterizedThreadStart(Calc));
        t1.Start(5.5);
        // ParameterizedThreadStart  인자 전달 & Start 인자 전달
        Thread t2 = new Thread(() =>Sum(10, 20, 30));
        t2.Start();
    }
}
```

Thread 강제 종료

- **Abort()** 메소드는 스레드를 파괴하는데 사용
  런타임은 **ThreadAbortException**을 발생시켜 스레드를 중단

```
public static void Run() {
    try {
        for (int i = 0; i <= 10; i++) {
            Thread.Sleep(500);
            Console.WriteLine(i);
        }
    } catch (ThreadAbortException e) {
        Console.WriteLine("ThreadAbortException");
    } finally { }
}
static void Main(string[] args) {
    Thread t = new Thread(Run);
    t.Start();
    Thread.Sleep(2000);
    Console.WriteLine("Mainthread aborts the child thread");
    t.Abort();
}
```

```
0
1
2
Mainthread aborts the child thread
ThreadAbortException
```

Thread 종료 대기

- **Join()** 메소드를 사용하여 해당 스레드가 종료될 때까지 호출 스레드를 차단

```
public static void Run() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine(Thread.CurrentThread.Name + " : " + i);
        Thread.Sleep(500);
    }
}
static void Main(string[] args) {
    Thread t1 = new Thread(Run);
    t1.Name = "Thread1";
    Thread t2 = new Thread(Run);
    t2.Name = "Thread2";
    t1.Start();
    t2.Start();
    t1.Join(); // 메인 스레드에서 t1이 종료될 때까지 대기
    t2.Join(); // 메인 스레드에서 t2가 종료될 때까지 대기
    Console.WriteLine("MainThread : done"); // 모든 스레드가 끝난 후 출력
}
```

```
Thread1 : 0
Thread2 : 0
Thread2 : 1
Thread1 : 1
Thread2 : 2
Thread1 : 2
Thread1 : 3
Thread2 : 3
Thread1 : 4
Thread2 : 4
MainThread : done
```

Background Thread

BackgroundThread

- Background Thread는 메인 스레드가 종료되면 바로 프로세스를 종료 스레드를 생성한 후 Start()를 실행하기 전에 IsBackground 속성을 true로 지정함
- IsBackground 속성의 기본값이 false인 Foreground Thread는 메인 스레드가 종료하더라도 살아있음

```
// Foreground 스레드
Thread t1 = new Thread(new ThreadStart(Run));
t1.Start();

// Background 스레드
Thread t2 = new Thread(new ThreadStart(Run));
t2.IsBackground = true;
t2.Start();
```

Thread

```
public static void Run() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine(Thread.CurrentThread.Name + " : " + i);
        Thread.Sleep(500);
    }
}
static void Main(string[] args) {
    Thread t1 = new Thread(Run);
    t1.Name = "Thread1";
    t1.IsBackground = true;
    Thread t2 = new Thread(Run);
    t2.Name = "Thread2";
    t2.IsBackground = true;
    t1.Start();
    t2.Start();
    t1.Join(); // 메인 스레드에서 t1이 종료될 때까지 대기
    t2.Join(); // 메인 스레드에서 t2이 종료될 때까지 대기
    for (int i = 0; i < 3; i++) {
        Console.WriteLine("MainThread : " + i);
        Thread.Sleep(10);
    }
}
```

```
Thread1 : 0
Thread2 : 0
Thread2 : 1
Thread1 : 1
Thread2 : 2
Thread1 : 2
Thread1 : 3
Thread2 : 3
Thread1 : 4
Thread2 : 4
MainThread : 0
MainThread : 1
MainThread : 2
```

Thread

```
public static void Run() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine(Thread.CurrentThread.Name + " : " + i);
        Thread.Sleep(500);
    }
}
static void Main(string[] args) {
    Thread t1 = new Thread(Run);
    t1.Name = "Thread1";
    t1.IsBackground = true; // 메인 스레드가 종료하면 배경 스레드 바로 종료
    Thread t2 = new Thread(Run);
    t2.Name = "Thread2";
    t2.IsBackground = true; // 메인 스레디가 종료하면 배경 스레드 바로 종료
    t1.Start();
    t2.Start();
    //t1.Join(); // 메인 스레드에서 t1이 종료될 때까지 대기하지 않는다면
    //t2.Join(); // 메인 스레드에서 t2가 종료될 때까지 대기하지 않는다면
    for (int i = 0; i < 3; i++) {
        Console.WriteLine("MainThread : " + i);
        Thread.Sleep(10);
    }
}
```

```
MainThread : 0
Thread1 : 0
Thread2 : 0
MainThread : 1
MainThread : 2
```

ThreadPool

- ThreadPool은 CLR에서 유지 관리하는 스레드 풀
- Thread 클래스를 이용하여 스레드를 하나씩 만들어 사용하는 것이 아니라, 이미 존재하는 스레드풀에서 사용가능한 작업 스레드를 할당 받아 사용하는 방식
- 이는 다수의 스레드를 계속 만들어 사용하는 것보다 효율적
- 스레드풀에 있는 스레드를 사용하기 위해서
  - ✓ ThreadPool 클래스
  - ✓ Asynchronous delegate
  - ✓ BackgroundWorker 클래스
  - ✓ Task Parallel Library (TPL)

ThreadPool 클래스

- ThreadPool.QueueUserWorkItem()를 사용하여 실행하고자 하는 메소드 델리게이트를 지정하면 시스템 풀에서 스레드를 할당하여 실행
- 이 방식은 실행되는 메소드로부터 리턴 값을 돌려받을 필요가 없는 곳에 주로 사용

```
public static void Calculate(object radius) {
    double r;
    if (radius==null) r = 1.0;
    else r = (double)radius;
    Console.WriteLine("r={0}, area={1}", r, r * r * 3.14);
}
static void Main(string[] args) {
    ThreadPool.QueueUserWorkItem(Calculate); // radius = null
    ThreadPool.QueueUserWorkItem(Calculate, 5.5);
    ThreadPool.QueueUserWorkItem(Calculate, 10.5);
    Console.ReadLine(); // 메인 스레드를 종료하지 않기 위해
}
```

```
r=10.5, area=346.185
r=1, area=3.14
r=5.5, area=94.985
```

Asynchronous Delegate

- .NET Core에서는 더 이상 지원하지 않음
- 메소드 대리자의 BeginInvoke()를 사용하여 스레드 작업을 시작, EndInvoke()를 사용하여 해당 작업이 끝날 때까지 기다려서 리턴 값을 넘겨 받음

```
delegate int WorkDelegate(int arg1, int arg2);
static int GetArea(int width, int height) {
    return width * height;
}
static void Main(string[] args) { // .NET Core에서 동작 안됨
    WorkDelegate work = GetArea;
    Console.WriteLine("Starting BeginInvoke");
    IAsyncResult result = work.BeginInvoke(4, 5, null, null);
    Console.WriteLine("Waiting on work in main…");
    int area = work.EndInvoke(result);
    Console.WriteLine("Area=" + area);
}
```

Asynchronous Delegate

- .NET Core에서는 IAsyncResult와 BeginInvoke/EndInvoke 대신 Task.Run과 del.Invoke를 사용

```
delegate int WorkDelegate(int arg1, int arg2);
static int GetArea(int width, int height) {
    return width * height;
}
static async void Main(string[] args) {
    WorkDelegate work = GetArea;
    Console.WriteLine("Starting with Task.Run instead of BeginInvoke");
    var workTask = Task.Run(() => work.Invoke(4, 5));
    Console.WriteLine("Waiting on work in main…");
    var area = await workTask; // 작업이 끝나기를 기다림
    Console.WriteLine("Area=" + area);
}
```

Task 클래스

- .NET4.0 부터 도입된 Task 클래스는 스레드 풀로부터 스레드를 가져와 비동기 작업을 실행
- .NET4.0 이전 ThreadPool.QueueUserWorkItem()와 같은 기능을 제공하지만, 보다 빠르고 유연함
- Task.Factory.StartNew()를 사용하여 실행하고자 하는 메소드에 대한 델리게이트를 지정
- StartNew()는 스레드를 생성과 동시에 실행하는 방식이고, 만약 시작을 하지 않고 Task 객체를 만들기 위해서는 Task() 생성자를 사용하여 메소드 델리게이트를 지정
- Non-Generic 타입인 Task 클래스는 ThreadPool.QueueUserWorkItem()과 같이 리턴값을 쉽게 돌려 받지 못함 Asynchronous Delegate와 같이 리턴값을 돌려 받기 위해서는 Task<T> 클래스를 사용

Task 클래스

```
static void Calculate(object radius) {
    double r;
    if (radius==null) r = 1.0;
    else r = (double)radius;
    Console.WriteLine("r={0}, area={1}", r, r * r * 3.14);
}
static void Run() {
    for (int i = 0; i < 5; i++) {
        Console.WriteLine("i={0}", i);
        Thread.Sleep(100);
    }
}
static void Main(string[] args) {
    _ = Task.Factory.StartNew(Calculate, null);
    _ = Task.Factory.StartNew(Calculate, 5.5);
    var task1 = new Task(Run); // create
    task1.Start(); // task start
    var task2 = Task.Factory.StartNew(() => Run()); // create & start
    var task3 = Task.Factory.StartNew(() => Run());
    var task4 = Task.Run(() => Run()); // run task
    Task.WaitAll(task1, task2, task3, task4);
    var task = Task.Factory.StartNew(() =>
    {
            Thread.Sleep(1000);
            return "Task Result";
    });
    Console.WriteLine("task=" + task.Result);
}
```

```
r=5.5, area=94.985
r=1, area=3.14
i=0
i=0
i=0
i=0
i=1
i=1
i=1
i=1
i=2
i=2
i=2
i=2
i=3
i=3
i=3
i=3
i=4
i=4
i=4
i=4
task=Task Result
```

Task<T> 클래스

```
static double GetArea(double width, double height) {
    return width * height;
}
static int Sum(int i, int j, int k) {
    return i + j + k;
}
static void Main(string[] args) {
    // Task<T>를 이용하여 스레드 생성과 시작
    var task5 = Task.Factory.StartNew<double>(() => GetArea(4.5, 5.5));
    var task6 = Task.Factory.StartNew<int>(() => Sum(3, 4, 5));
    // 메인스레드에서 다른 작업 실행
    Thread.Sleep(1000);
    // 스레드 결과 리턴. 스레드가 계속 실행중이면
    // 이곳에서 끝날 때까지 대기함
    var result = task5.Result;
    Console.WriteLine("task5 Result={0}", result);
    result = task6.Result;
    Console.WriteLine("task6 Result={0}", result);
}
```

```
task5 Result=24.75
task6 Result=12
```

BackgroundWorker 클래스

- BackgroundWorker 클래스는 스레드 풀에서
  작업 스레드(Worker Thread)를 할당 받아 작업을 배경에서 실행
- System.ComponentModel namespace
- Event-based Asynchronous Pattern을 구현한 클래스
- DoWorkEventHandler를 통해 작업할 내용을 지정하고,
  RunWorkerAsync() 메소드를 호출하여 작업을 시작
- WindowForm에서는 UI 스레드와는 별도로
  BackgroundWorker 스레드를 이용하면 별도 작업을 수행할 수 있음
- BackgroundWorker 이벤트
  - ✓ DoWorker 이벤트 – 실제 작업할 내용을 지정하는 이벤트
  - ✓ ProgressChanged 이벤트 – DoWorker의 진척 사항을 전달
  - ✓ RunWorkerCompleted 이벤트 – 작업 완료 이벤트

BackgroundWorker 클래스

```
static BackgroundWorker worker = new BackgroundWorker();
static void Worker_DoWork(object sender, DoWorkEventArgs e) {
    int i = 0;
    for (i = 1; i < 10; i++) {
        Thread.Sleep(1000);
        worker.ReportProgress(10 * i); // progress %
    }
    e.Result = i;
}
static void Worker_ProgressChanged(object sender, ProgressChangedEventArgs e) {
    Console.WriteLine("Progress : {0} %", e.ProgressPercentage);
}
static void Worker_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e) {
    Console.WriteLine("Value of i = " + e.Result);
    Console.WriteLine("RunWorkerCompleted");
}
static void Main(string[] args) {
    worker.WorkerReportsProgress = true;
    worker.WorkerSupportsCancellation = true;
    worker.DoWork += Worker_DoWork;
    worker.ProgressChanged += Worker_ProgressChanged;
    worker.RunWorkerCompleted += Worker_RunWorkerCompleted;
    Console.WriteLine("Start background worker");
    worker.RunWorkerAsync(); // start background worker
    Console.ReadKey(); // 메인스레드를 종료하지 않기 위해
}
```

```
Starting Background worker...
Progress : 10 %
Progress : 20 %
Progress : 30 %
Progress : 40 %
Progress : 50 %
Progress : 60 %
Progress : 70 %
Progress : 80 %
Progress : 90 %
Value Of i = 10
RunWorkerCompleted
```

Task.Run

```
static async void Main(string[] args) {
    // Task.Run
    var progressHandler = new Progress<int>(value =>
    {
        Console.WriteLine("Progress : {0} %", value);
    });
    var progress = progressHandler as IProgress<int>;
    await Task.Run(() =>
    {
        for (int i = 0; i <= 100; ++i) {
            if (progress != null)
                progress.Report(i);
            Thread.Sleep(100);
        }
    });
    Console.WriteLine("Task.Run Completed");
}
```

```
Progress : 0 %
Progress : 1 %
Progress : 2 %
...
Progress : 90 %
Progress : 91 %
Progress : 92 %
Progress : 93 %
Progress : 94 %
Progress : 95 %
Progress : 96 %
Progress : 97 %
Progress : 98 %
Progress : 99 %
Progress : 100 %
Task.Run Completed
```

Window Forms Thread 사용

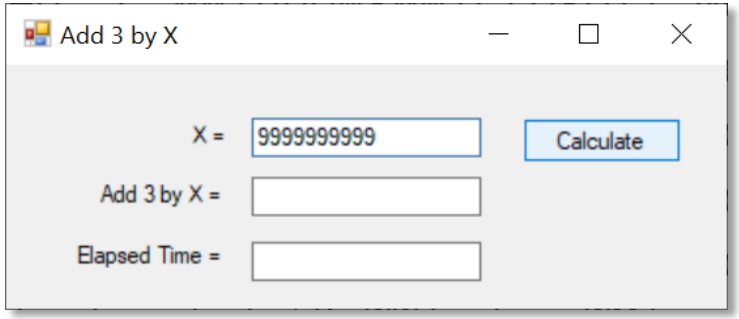
- .NET에서 UI Application은 Window Forms나 WPF (Window Presentation Foundation)을 사용
- UI Thread는 WinForm이나 WPF의 UI 컨트롤을 생성하고 이 컨트롤의 윈도우 핸들을 소유한 스레드
- UI Thread만이 해당 UI 객체를 액세스할 수 있다는 Thread Affinity (스레드 선호도) 규칙을 지키도록 설계됨
- Worker Thread (작업 스레드)는 UI 컨트롤을 갖지 않는 스레드
- WinForm이나 WPF는 하나의 UI Thread(메인 스레드)를 가지며, 여러 개의 Worker Thread를 가짐
- 하지만 필요한 경우 여러 개의 UI Thread 가질 수 있음

UI Thread

- Window Forms의 UI 컨트롤들은 Control 클래스로부터 파생된 클래스
- Control 클래스는 UI 컨트롤이 UI Thread에서 돌고 있는지를 체크하는 InvokeRequired 속성을 가짐
- 작업 스레드에서 UI 컨트롤을 접근하기 위해 Control 클래스의 Invoke()나 BeginInvoke() 메소드를 사용하여 UI Thread로 작업 요청을 보냄
- .NET4.0, C# 5.0 이상의 경우 Invoke()나 BeginInvoke() 메소드 대신 async, await, Task.Run을 사용하여 UI Thread와 별도의 작업 요청

UI Thread

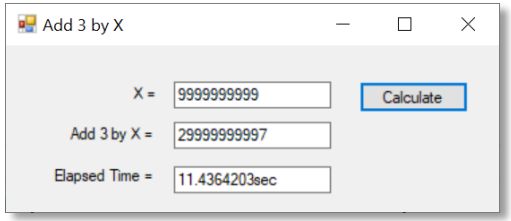
```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e) {
        long value = 0;
        if (long.TryParse(textBox1.Text, out value)) {
            textBox2.Text = Add3(value).ToString(); // 오래 걸리는 작업은 작업 종료 전까지 UI 멈춤 발생
– 계산하는 동안 MainThread가 UI 컨트롤 핸들링을 못함
        } else {
            textBox2.Text = "Wrong number";
        }
    }
    private long Add3(long value) {
        long result = 0;
        for (long i = 0; i < value; i++) {
            result += 3;
        }
        return result;
    }
}
```



UI Thread

- **async**는 해당 메소드가 **await**을 가지고 있음을 알려줌
- **await**는 작업이 종료될 때까지 기다렸다가 완료 후 다음 실행문 코드 **(await** 호출 이전의 스레드**)** 실행

```
private async void button1_ClickAsync(object sender, EventArgs e) {
        long value = 0;
        long result = 0;
        if (long.TryParse(textBox1.Text, out value)) {
            DateTime startTime = DateTime.Now;
            await Task.Run(() =>
            {
                result = Add3(value); // async thread 처리
            }); // await 작업이 종료될 때까지 대기함. UI Thread는 block되지 않음
            textBox2.Text = result.ToString();
            textBox3.Text = (DateTime.Now – startTime).TotalSeconds + “ sec”;
        } else {
            textBox2.Text = "Wrong number";
        }
    }
```



Thread Synchr

멀티스레드 프로그램 작성시 주의점

- 다수의 스레드가 공유 데이터에 동시에 접근하는 경우
  - ✓ 공유 데이터의 값에 예상치 못한 결과 발생 가능

스레드 동기화 (Thread Synchronization)

- 멀티스레드의 공유 데이터의 동시 접근 문제 해결책
  - ✓ 공유 데이터를 접근하는 모든 스레드들이 순차적으로 접근하도록 함
  - ✓ 한 스레드가 공유 데이터에 대한 작업을 끝낼 때까지 다른 스레드가 대기 하도록 함
- 이렇게 스레드 동기화를 구현한 메소드나 클래스를 Thread-Safe 하다고 함
- .NET의 많은 클래스들은 Thread-Safe하지 않음

스레드 동기화를 위한 .NET 클래스들

- lock 키워드는 특정 코드 블록(Critical Section)을 한번에 하나의 스레드만 실행할 수 있게 해줌
- Monitor 클래스는 lock과 같이 특정 코드 블록(Critical Section)을 배타적으로 lock 하는 기능을 가지고 있음
  한 Process 내에서만 사용 가능
- Mutex 클래스는 Monitor 클래스 같이 특정 코드 블록(Critical Section)을 배타적으로 lock 하는 기능을 가지고 있음
  해당 머신의 Process 간에서도 배타적 lock 가능
- Semaphore 클래스는 공유된 리소스를 지정된 수의 스레드들만 엑세스하는 것을 허용

스레드 동기화를 위한 .NET 클래스들

- SpinLock 클래스
- ReaderWriteLock 클래스
- AutoResetEvent 클래스
- ManualResetEvent
- CountdownEvent