

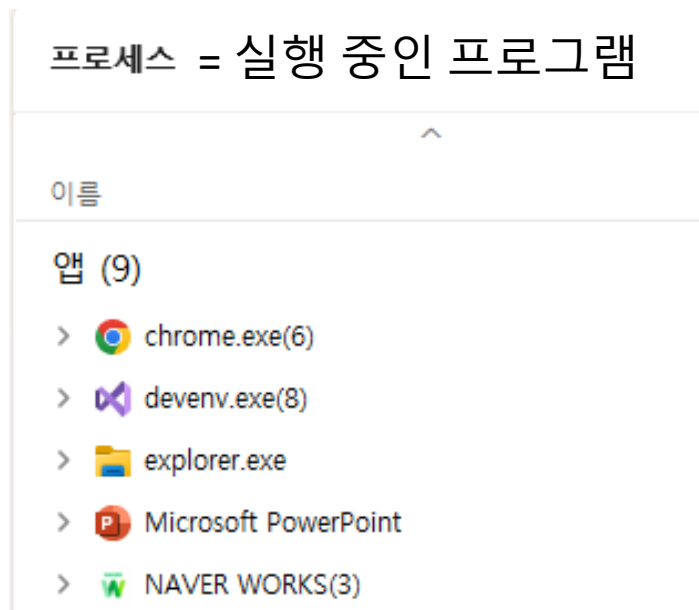
# 멀티스레드

# 용어 정리

- **프로그램** : 우리가 실행할 수 있는 .exe 같은 파일
- **프로세스(Process)** : 프로그램이 실행되어 메모리 상에 올라간 것.
  - 이 프로세스는 **하나 이상의 스레드(Thread)**를 포함할 수 있음.
- **스레드(Thread)** : 프로세스 내에서 실제로 **코드를 실행하는 작업 단위**
  - 하나의 프로세스는 최소 하나의 스레드를 갖고 있음.

# 스레드

- Thread
  - Process를 구성하는 작업 수행 주체



## CPU 11th Gen Intel(R) Core(TM) i5-1155G7 @ 2.50GHz



이용률	속도	기본 속도:	2.50GHz
20%	3.16GHz	소켓:	1
프로세스	스레드	코어:	4
291	3885	논리 프로세서:	8
작동 시간	핸들	가상화:	사용
8:01:17:06	146422	L1 캐시:	320KB
		L2 캐시:	5.0MB
		L3 캐시:	8.0MB

스레드

= 프로세스 내부에서 작동하는 기능  
최소 1개의 스레드를 가짐 (main)

하나의 프로세스가 여러 개의 스레드를  
갖는 것이 가능

CPU는 스레드 단위로 일 함

# 싱글 스레드

- 싱글 스레드
  - 하나의 작업만 순차적으로 처리하는 방식
- 특징
  - 한 번에 한 작업만 실행 가능
  - 이전 작업이 끝나야 다음 작업이 실행됨.
  - 코드 실행 흐름이 직선형(순차적)

# 멀티 스레드

- 멀티 스레드

- 여러 작업을 동시에 처리하는 방식

- 특징

- 두 개 이상의 작업을 병렬로 실행
  - 백그라운드 작업을 수행하면서도 UI가 멈추지 않음.
  - 처리 속도 향상 (CPU 효율 증가)

# 멀티스레드

CPU 시간 →

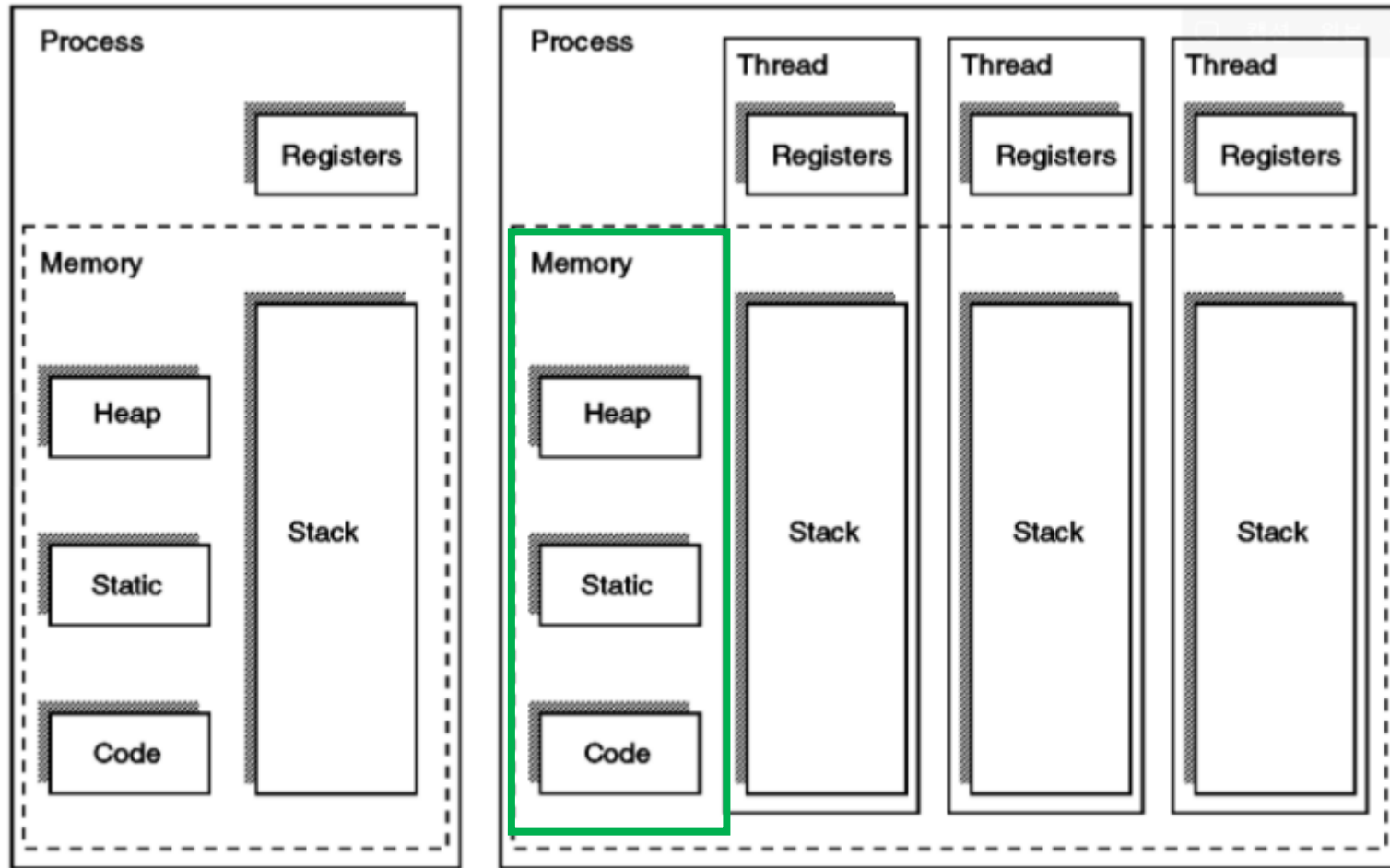


싱글코어 -> 단일 스레드



멀티코어 -> 멀티 스레드

# 멀티스레드



🌟 Heap, Static 두 가지는 스레드간 공유되지만, Stack은 안됨

# 멀티스레드

- 스레드는 함수 단위로 동작함
- 하나의 프로세스(프로그램)에 여러 개의 스레드를 만들면 OS(윈도우)가 스레드의 실행 순서를 정함
  - 의도적으로 스레드의 실행 순서를 제어할 수 없음
- 스레드의 소스코드가 아직 전부 실행되지 않았는데, OS가 다른 스레드를 실행하여 갑자기 다른 코드가 작동되는 것을 Context Switching 이라고 함
- 멀티스레드 방식은 단일 스레드 방식 대비 메모리 절약, 속도 개선과 같은 장점이 있음
- 현재는 윈도우, 맥, 안드로이드, iOS 등 대부분의 운영체제에서 동작하는 거의 모든 프로그램들은 멀티스레드로 동작함



# 멀티스레드

- Context Switching
  - 스레드의 소스코드가 아직 전부 실행되지 않았는데, OS가 다른 스레드를 실행하여 갑자기 다른 코드가 작동되는 것
- 동기 처리(Synchronous)
  - 단일 스레드 환경에서 소스코드가 순차적으로 실행되는 것
- 비동기 처리(Asynchronous)
  - 멀티 스레드 방식으로 Context Switching을 활용하여 마치 병렬처리가 되는 것처럼 소스코드가 실행되는 것

# 멀티스레드

- .NET Framework에서 제공하는 멀티스레드 클래스

## 1. BackgroundWorker

- WinForm 또는 WPF의 UI 스레드와 상관없이 코드를 실행하는 것이 목적
- 파일 다운로드, 데이터베이스 조회, 프로그램스 바 갱신 등에 활용

## 2. Thread

- 일반적으로 스레드를 직접 관리할 때 사용하는 클래스
- BackgroundWorker 대비 세밀한 제어가 가능하지만, 스레드 생명 주기, 동기화 등을 직접 관리해야 하기 때문에 복잡성이 증가함
- 시작(Start), 중지(Abort), 일시 중지(Suspend), 재개(Resume) 기능 지원


# BackgroundWorker

- DoWork
  - BackgroundWorker가 UI 스레드와 별개로 수행할 메소드를 지정
- ProgressChanged
  - DoWork()로 지정된 메소드에서 ReportProgress() 메소드가 호출되면 반복적으로 실행되는 메소드를 지정 `this.worker.ReportProgress(0);`
  - ReportProgress()는 0~100 사이의 값을 입력 받을 수 있음
- RunWorkerCompleted
  - DoWork() 메소드가 끝나면 1회 동작하는 메소드를 지정
- RunWorkerAsync()
  - BackgroundWorker를 실행시키는 메소드 `this.worker.RunWorkerAsync();`

# BackgroundWorker

```
this.worker = new BackgroundWorker();  
// BackgroundWorker의 ReportProgress() 메소드 활용 여부, 보통 true  
this.worker.WorkerReportsProgress = true;  
// CancelAsync()로 BackgroundWorker를 멈출 수 있게 할지, 보통 true  
this.worker.WorkerSupportsCancellation = true;  
  
// BackgroundWorker가 UI스레드와 별개로 수행할 메소드  
this.worker.DoWork += new DoWorkEventHandler(Worker_DoWork);  
// ReportProgress() 메소드가 수행될때 실행될 메소드  
this.worker.ProgressChanged += new ProgressChangedEventHandler(Worker_ProgressChanged);  
// ReportProgress()가 100으로 호출되면 마지막에 한 번 실행되는 메소드  
this.worker.RunWorkerCompleted += new RunWorkerCompletedEventHandler(Worker_Complete);  
  
void Worker_DoWork(object sender, DoWorkEventArgs e)  
void Worker_ProgressChanged(object sender, ProgressChangedEventArgs e)  
void Worker_Complete(object sender, EventArgs e)
```

메소드 이름

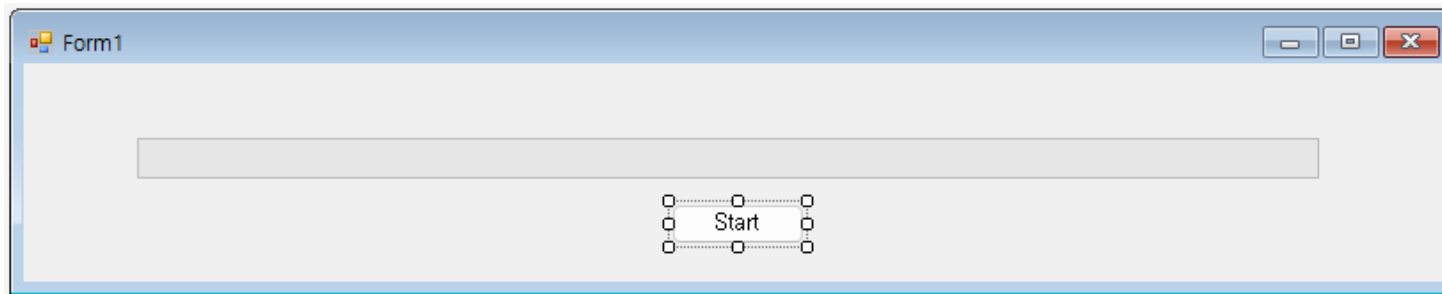


# BackgroundWorker

- 프로그래스 바를 만들 때, while문을 사용하여 바를 채우게 되면 UI 스레드가 바쁠 경우 응답 없음이 되고, 순식간에 바가 가득 차는 문제가 발생

```
while (true)
{
    progressBar1.Value += 1;

    if (progressBar1.Value >= 100) break;
}
```



# 실습. BackgroundWorker

- BackgroundWorker를 사용하여 UI 스레드와 별개로 프로그래스 바를 작동 시키기
  1. 프로그래스 바를 만들고 값의 범위를 0~100으로 설정
    - <https://learn.microsoft.com/ko-kr/dotnet/api/system.windows.forms.progressbar?view=windowsdesktop-8.0>
  2. DoWork에 할당된 메소드는 반복문을 30ms 마다 총 101회 돌면서 0 부터 100까지 ReportProgress(int)를 호출
    - Thread.Sleep()을 사용하여 30ms의 시간차를 발생시킴
  3. ProgressChanged에 할당된 메소드는 프로그래스 바의 Value를 1씩 증가 시킴
  4. RunWorkerCompleted에 할당된 메소드는 "완료됨" 메시지 박스를 호출
  5. Start 버튼을 클릭시 BackgroundWorker가 동작

# Thread

- using System.Threading 선언

```
1 reference
public Form1()
{
    InitializeComponent();

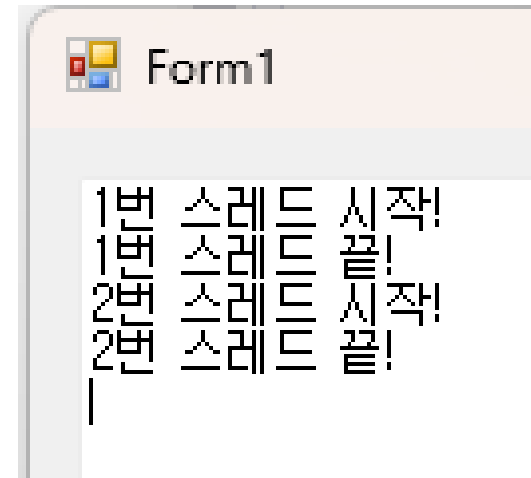
    textBox_print.Text += "1번 스레드 시작!\r\n";

    var new_thread = new Thread(new ThreadStart(MyThread));
    new_thread.Start();

    textBox_print.Text += "1번 스레드 끝!\r\n";
}
```

```
1 reference
void MyThread()
{
    textBox_print.Text += "2번 스레드 시작!\r\n";
    Thread.Sleep(1000);
    textBox_print.Text += "2번 스레드 끝!\r\n";
}
```

함수 단위로 스레드를 생성



# Thread

```
static int sharedData = 0;
```

```
1 reference
public Form1()
{
    InitializeComponent();

    Thread thread1 = new Thread(UpdateData1);
    Thread thread2 = new Thread(UpdateData2);

    thread1.Start();
    thread2.Start();

    // join: 메인 스레드는 이 스레드가 끝날때까지 대기
    thread1.Join();
    thread2.Join();
}
```

```
1 reference
void UpdateData1()
{
    for(int i = 0; i < 10; i++)
    {
        sharedData++;
        textBox_print.Text += $"1: " + sharedData + "\r\n";
    }
}

1 reference
void UpdateData2()
{
    for (int i = 0; i < 10; i++)
    {
        sharedData++;
        textBox_print.Text += $"2: " + sharedData + "\r\n";
    }
}
```

 Form1

```
2: 2
1: 3
1: 4
1: 5
1: 6
1: 7
2: 9
2: 12
2: 14
2: 15
2: 16
2: 17
2: 18
2: 19
2: 20
```



# Thread - lock

- 컨텍스트 스위칭이 일어나서는 안되는 코드를 보호하는 기능
- lock으로 감싸진 코드는 실행 중에 컨텍스트 스위칭이 일어나지 않음
- lock으로 너무 넓은 범위를 감싸게 되면 멀티스레딩의 의미가 없어지기 때문에 적재 적소에 사용하는 것이 필요
  - 메소드 전체를 lock 으로 지정한다면 일반 메소드와 다를 것이 없어짐
- lock을 사용하기 위해서는 스레드간 lock 정보를 공유해야 하기 때문에 **static** 영역에 lock 상태를 담고 있는 변수를 선언 해야함

# 멀티스레드 - lock

```
static int sharedData = 0;
static object lockObject = new object();
```

```
1 reference
public Form1()
{
    InitializeComponent();

    Thread thread1 = new Thread(UpdateData1);
    Thread thread2 = new Thread(UpdateData2);

    thread1.Start();
    thread2.Start();

    // join: 메인 스레드는 이 스레드가 끝날때까지 대기
    thread1.Join();
    thread2.Join();
}
```

```
1 reference
void UpdateData1()
{
    for(int i = 0; i < 10; i++)
    {
        lock (lockObject)
        {
            sharedData++;
            textBox_print.Text += $"1: " + sharedData + "\r\n";
        }
    }
}
```

```
1 reference
void UpdateData2()
{
    for (int i = 0; i < 10; i++)
    {
        lock (lockObject)
        {
            sharedData++;
            textBox_print.Text += $"2: " + sharedData + "\r\n";
        }
    }
}
```

Form1

```
1: 1
1: 2
1: 3
1: 4
1: 5
1: 6
1: 7
1: 8
1: 9
1: 10
2: 11
2: 12
2: 13
2: 14
2: 15
2: 16
2: 17
2: 18
2: 19
2: 20
```

# 멀티스레드 - 주의사항!

- WinForm에서 멀티스레드를 사용하다 보면 "다른 스레드가 이 개체를 소유하고 있어 호출한 스레드가 해당 객체에 액세스 할 수 없습니다." 라는 예러가 발생 할 수 있음
- WinForm의 UI는 Main Thread가 소유하고 있기 때문에 다른 스레드에서 접근할 수 없음
- 해결 방법
  1. Dispatcher  
우선순위 큐, UI 컨트롤들은 하나의 Dispatcher를 가지고 있으며 MainThread가 소유
  2. SynchronizationContext  
스레드간의 통신을 담당

# 실습. 멀티스레드 레이스

- Thread 및 Thread.Sleep 메소드를 사용하여 아래 레이스 경기 코드를 제작
  1. 총 5명의 참가자(차량)가 동시에 경주 시작. (차량마다 스레드 생성)
  2. 각 차량은 랜덤한 시간 간격으로 전진함. (0.1초 ~ 1초)
  3. 차량이 결승선에 도달하면 차량 이름 및 시간을 출력
  4. 모든 차량이 결승선에 도달하면 레이스 종료 메시지 출력 및 경기 종료
  5. (Hint) DateTime, TimeSpan, List<Thread> 활용.

결과화면 캡처& GitHub Repo. URL을 슬랙 댓글로 제출

- 스레드에서 안전하게 UI 호출하기 (<https://learn.microsoft.com/ko-kr/dotnet/desktop/winforms/controls/how-to-make-thread-safe-calls?view=netdesktop-8.0>)

# Task, async, await

- 간편하게 멀티스레드를 구현하기 위한 기능
- Task, Task<T>
  - 비동기 작업을 나타내는 객체
  - 비동기 작업의 완료, 실행, 또는 함수 실행 결과를 반환하는 역할을 수행
- async
  - 메소드 선언 앞에 async가 있다면 비동기로 실행이 가능한 메소드
  - void, Task 또는 Task<T> 만 반환할 수 있으나, void 반환 시 호출하는 쪽에서 비동기 처리가 불가능
  - await을 통해 호출되어야 비동기로 작동함
- await
  - 작업이 끝나기를 기다리지만 스레드를 멈추지는 않음
  - Task 또는 Task<T> 반환 하는 메소드만 기다릴 수 있음
  - async로 선언된 메소드에서만 사용 가능

# Task, async, await

- async로 선언되고 Task를 반환하는 메소드를 async로 선언된 다른 메소드에서 await으로 호출해서 사용하는 것이 기본적인 사용법

```
async Task<string> AsyncOperation3()
{
    string word = "";
    for (int i = 0; i < 12345678; i++)
    {
        word = i.ToString();
    }

    return word;
}
```

```
private async void button_run_Click(object sender, EventArgs e)
{
    string result = await AsyncOperation3();
}
```

- \* await을 사용하기 위해 호출하는 쪽도 async로 선언돼야 함
- \* AsyncOperation3()가 끝나기를 기다리지만 스레드를 멈추지는 않음

# Task, async, await

- async로 선언된 메소드 내부에서 Task 객체를 생성하고 바로 await을 사용하여 비동기 처리를 하는 것도 가능

```
private async void button_run_Click(object sender, EventArgs e)
{
    // Task 인스턴스 생성 및 실행
    Task task = new Task(() =>
    {
        // 다른 스레드 영역
        Thread.Sleep(1000);
    });
    task.Start();
    await task; // task 종료까지 대기
}
```

# Task, async, await

- async로 선언되지 않은 메소드도 Task를 반환한다면 await으로 호출하여 비동기 처리 가능

```
Task<int> AsyncOperation2()
{
    return Task.Run(() =>
    {
        // 새 스레드에서 동작할 코드
        Thread.Sleep(1000); // 시간 만큼 스레드 멈춤
        return 99;
    });
}
```

```
async void button_run_Click(object sender, EventArgs e)
{
    int result = await AsyncOperation2();
    textBox_print.Text += $"A02 Done: {result}\r\n";
}
```



# Task, async, await

- async로 선언된 메소드는 비동기 방식으로 호출될 가능성이 있으므로 UI 컨트롤 수정은 Invoke를 통해서 처리

```
// 비동기 처리 중이어서 Invoke가 필요한 상황인지 판단
if (textBox_print.InvokeRequired)
{
    // Action은 입출력이 없는 delegate
    // 주로 람다식과 함께 간편하게 delegate를 표현하는데 사용
    this.Invoke((Action)(() =>
    {
        // 비동기 처리되고 있으므로 UI 업데이트를 위해 Invoke 사용
        textBox_print.Text += "Invoke Action\r\n";
    }));
}
else
{
    textBox_print.Text += "Invoke Action\r\n";
}
```

# Task, async, await

- Task.Delay()를 사용하여 await 과 함께 사용할 때 스레드를 멈추지 않고 대기하는 것이 가능
  - 즉, UI 스레드나 다른 스레드의 동작을 차단하거나 방해하지 않음
- 주로 I/O 대기, 네트워크 요청 대기 등에서 활용

```
async Task<int> AsyncOperation1()  
{  
    // Delay(): await과 함께 사용시 스레드를 멈추지 않고 시간만큼 대기  
    await Task.Delay(1000); // 이 부분만 비동기 처리  
}
```

# 실습. **async, await**

- Task, async, await을 사용하여 아래 기능을 구현
- 버튼을 클릭하여 텍스트 파일을 선택하면, 파일의 내용을 텍스트박스에 표시해주는 프로그램
  1. 텍스트 파일을 비동기적으로 읽어오는 **ReadFileAsync** 메소드 작성
  2. StreamReader를 이용하여 파일의 내용을 불러오고, StreamReader에 내장된 **ReadToEndAsync()** 메소드를 사용하여 읽어오는 부분도 비동기처리
- Push 후 GitHub Repo. URL을 슬랙 댓글로 제출