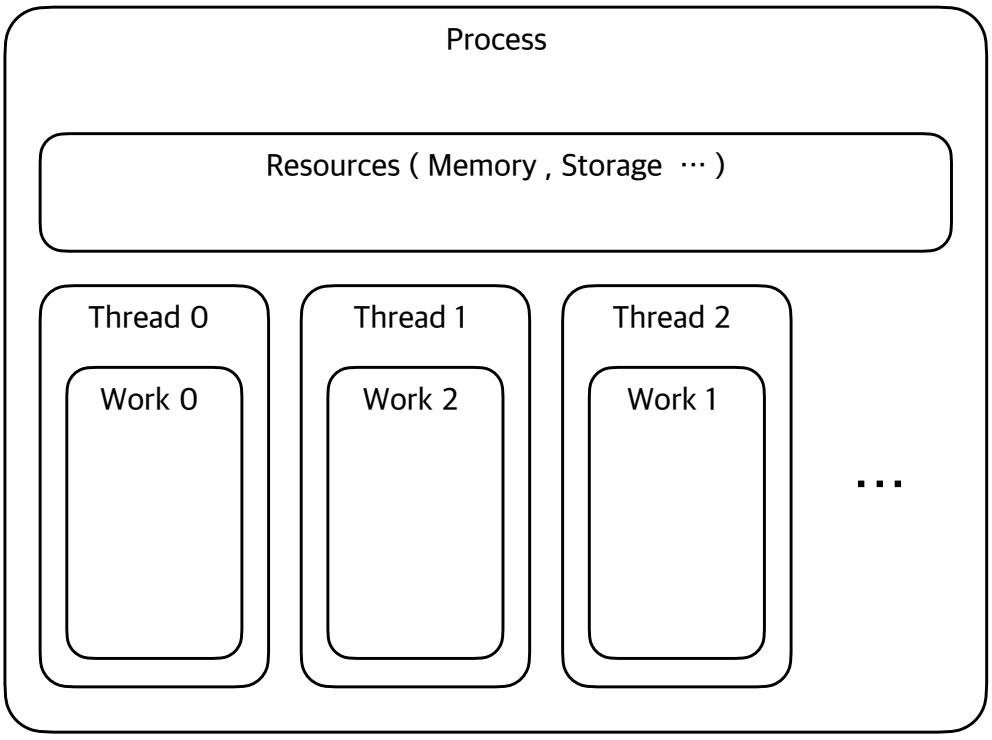
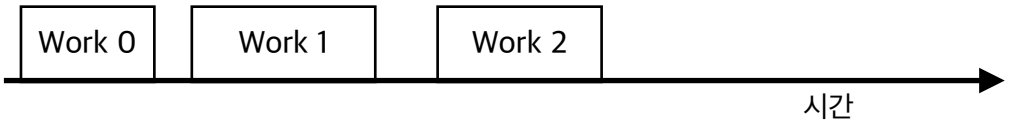
멀티태스킹-멀티쓰레딩

* 멀티쓰레딩



< 멀티쓰레드 프로그램 내에서의 쓰레드와 작업 구조>



<싱글쓰레드 프로그램에서의 작업 흐름>

<멀티쓰레드 프로그램에서의 작업 흐름 (best-case)>

- 한 프로세스 내에서 여러개의 작업 흐름을 실행하는 기술
- 멀티 코어를 활용하여 작업의 병렬 처리가 가능
- 큰 작업을 분할하여 빠른 시간내에 처리하기에 적합
- GUI 프로그램에서의 렌더링을 방해하지 않고 작업 처리가 가능하도록 하여, 사용자 경험을 개선

```
3 references
public class ThreadTest{
    3 references
    private string Name;
    2 references
    public ThreadTest(string name)
    {
        Name = name;
    }

    1 reference
    public void RunThread()
    {
        // run thread with work method
        Thread t = new Thread(MyWork);
        t.Start();
    }

    // Actual Job
    1 reference
    private void MyWork()
    {
        Console.WriteLine($"{Name}-Working..");
        Thread.Sleep(500);
        Console.WriteLine($"{Name}-WorkComplete!");
    }

}
```

```
onsoleApp1 > Program.cs
1   // See https://aka.ms/new-console-template for more information
2   Console.WriteLine("Hello, World!");
3
4   new Tests().TestThread();
```

그렇다면 쓰레드는 많을수록 좋은가?

- 각각의 쓰레드들이 작업을 마치는 타이밍이 다르기 때문에 이들 작업 처리 결과의 동기화가 필요

```
Hello, World!
Thread-0-Working..
Thread-1-Working..
Thread-2-Working..
Thread-3-Working..
Thread-4-Working..
Threads are Running ..
Thread-1-WorkComplete!
Thread-2-WorkComplete!
Thread-0-WorkComplete!
Thread-4-WorkComplete!
Thread-3-WorkComplete!
```

<실행이 line-by-line 순으로 일어나지 않는다>

- 동시 접근이 허용되지 않은 리소스에 여러 쓰레드가 접근하는 경우 예측할 수 없는 오류 발생 가능

* Race Condition :
  두 개 이상의 스레드가 하나의 자원(변수) 에 접근하며, 작업 타이밍이 다를 때 예측할 수 없는 작업 결과를 만드는 것.
* Critical Section :
  Race Condition 이 발생하지 않도록 보호해야 하는 영역

-> 특히 메모리에 동시 접근하는 경우를 방지하기 위해 Mutex, Semaphore 등을 사용

```
int incrTest = 0;
1 reference
public void RunCriticalSectionTest()
{
    for(int i = 0; i < 5 ; i++)
    {
        Thread t = new Thread(WorkWithSharedMem);
        t.Start();
    }
}

1 reference
private void WorkWithSharedMem()
{
    int myNum = threadNum;
    threadNum++;
    while(incrTest < 100)
    {
        Thread.Sleep(1);
        Console.WriteLine($"{myNum}-Working..{incrTest}");
        incrTest++;
    }
}
```

```
3-Working..80
0-Working..82
4-Working..82
2-Working..84
1-Working..85
4-Working..85
2-Working..85
3-Working..85
0-Working..89
1-Working..90
3-Working..91
4-Working..92
2-Working..93
0-Working..94
1-Working..95
4-Working..95
3-Working..97
2-Working..97
0-Working..97
1-Working..100
2-Working..100
4-Working..100
3-Working..102
```

< Race Condition 이 발생하는 코드와 결과 >

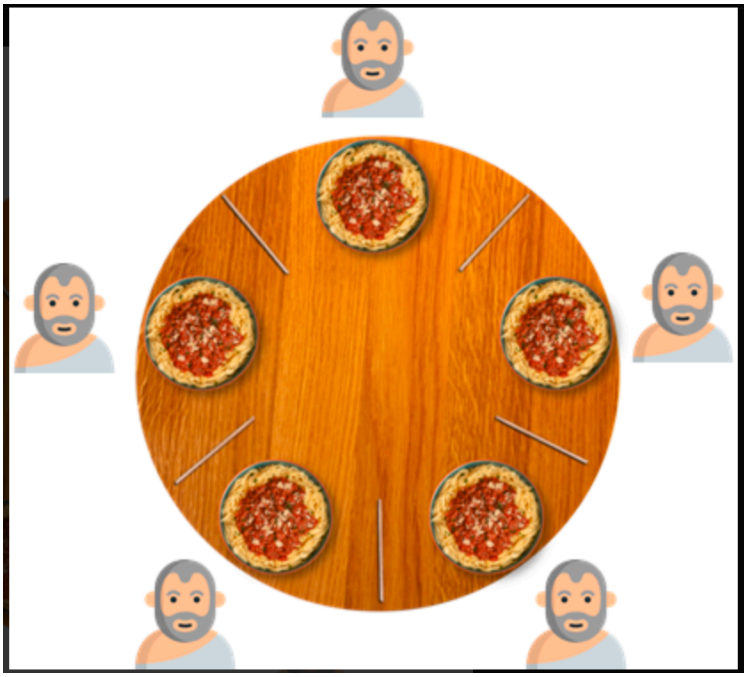
- Critical Section 을 보호하기 위한 처리는 결국 그동안 다른 쓰레드가 작업을 멈추고 대기 하게됨

- 쓰레드 수가 쓰레드를 처리 할 수 있는 코어 수 보다 많을때, 쓰레드의 작업을 전환하는 '컨텍스트 스위칭' 비용 발생

- 대부분의 응용 프레임워크나 엔진들은 대부분의 라이브러리를 메인 쓰레드에서만 동작하도록 제한하므로 멀티쓰레드를 통해 처리할 작업 역시 제한된다

- 자원을 보호하는 코드로 인해 '데드락' 현상이 발생 할 수 있다.

데드락의 예 - 식사하는 철학자들



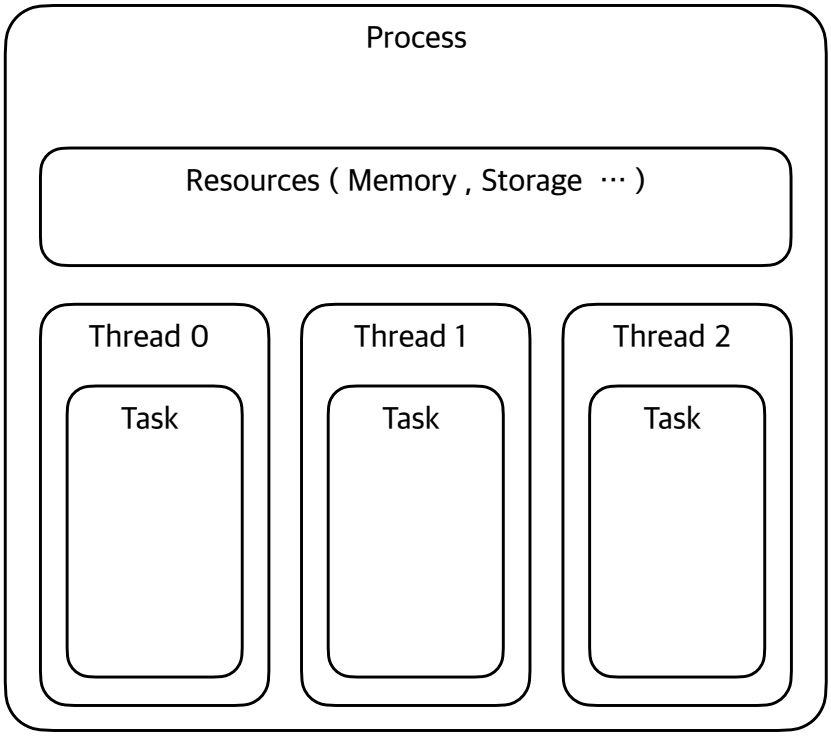
철학자는

1. 왼쪽 **포크**가 있다면 왼쪽 **포크**를 든다
2. 오른쪽 **포크**가 있다면 오른쪽 **포크**를 든다
3. 양손에 **포크**를 들었다면 **식사**를 한다
4. 오른쪽 **포크**를 내려놓는다
5. 왼쪽 **포크**를 내려놓는다

*** As code**

```
if( resource A available )
        lock(A)
        if ( resource B available )
                lock(B)
                Process(A , B )
                unlock(B)
                unlock(A)
```

* 멀티 태스킹 (with .NET Task)



- 미리 생성되어있는 쓰레드 (쓰레드 풀) 에 특정 작업을 하게 하는 방법
- '쓰레드 풀' 중, 구동환경에서 반환된 쓰레드가 호출된 태스크를 수행
- 멀티쓰레드 상황에서 발생할 수 있는 문제는 동일하게 발생

```
public class TaskTest
{
    3 references
    private string Name;
    2 references
    public TaskTest(string name)
    {
        Name = name;
    }

    1 reference
    public void RunTask()
    {
        Task.Run(MyWork);
    }

    1 reference
    public Task RunTaskAndGetHandle()
    {
        return Task.Run( MyWork);
    }

    2 references
    private async Task MyWork()
    {
        Console.WriteLine($"{Name}-Working..");
        await Task.Delay(500);
        Console.WriteLine($"{Name}-WorkComplete!");
    }
}
```

- async 키워드로 비동기 작업을 선언하고,
- await 키워드로 비동기 작업을 대기 함

- 비동기 작업으로 선언되었으나 호출 코드에서 대기 할 필요가 없는경우,
  await 하지 않아도 됩니다