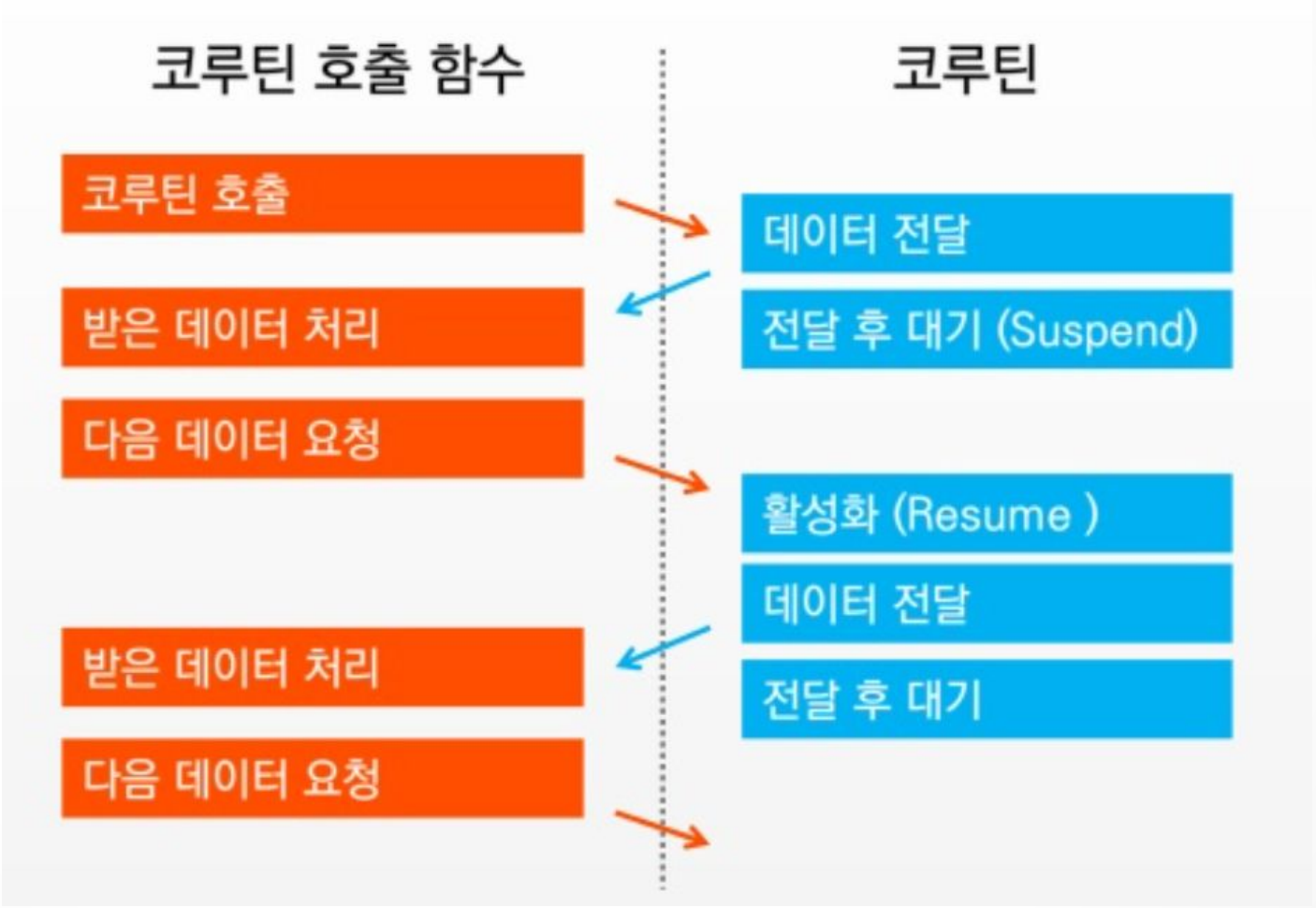
유니티 코루틴 & 멀티 스레드

이준

코루틴이란?

- 유니티에서 동시적인 프로그래밍을 가능하게하는 기술
  - 유니티만의 기술은 아니고 OS 및 프로그래밍 연구자들의 기술을 유니티가 수용
  - 유니티는 싱글 스레드로 만들어진 시스템이기 때문에 블록킹 현상이 발생하는 상황을 해결하기 위해서 코루틴을 적용함
- 서브루틴
  - 진입지점이 하나인 함수, 함수 선언부분에서 진입함
  - Return 구문으로 반환되면 다시 처름 부터 진입 해야함
- 코루틴
  - 집입 지점이 여러개인 함수
  - Yield return 구문으로 반환되고 다시 들어올 수 있음
  - 유니티에서는 IEnumerator 인터페이스를 반환하도록 설정함

코루틴이란?


코루틴 호출 함수
코루틴
코루틴 호출
데이터 전달
받은 데이터 처리
전달 후 대기 (Suspend)
다음 데이터 요청
활성화 (Resume)
데이터 전달
받은 데이터 처리
전달 후 대기
다음 데이터 요청

코루틴이란?

- IEnumerable 콜렉션
  - Foreach 구문등에서 객체를 한개 한개 넘겨주는 역할을 수행
  - 먼저 처음 물건을 넘겨준다. 그다음 물건을… 넘겨주는 쪽이 몇번째 인지 기억을 해야함!

코루틴이란?

- IEnumerable 콜렉션
  - 넘겨주는 쪽이 누가 몇번째까지 받았는지 모두 기억할 수 있을까??

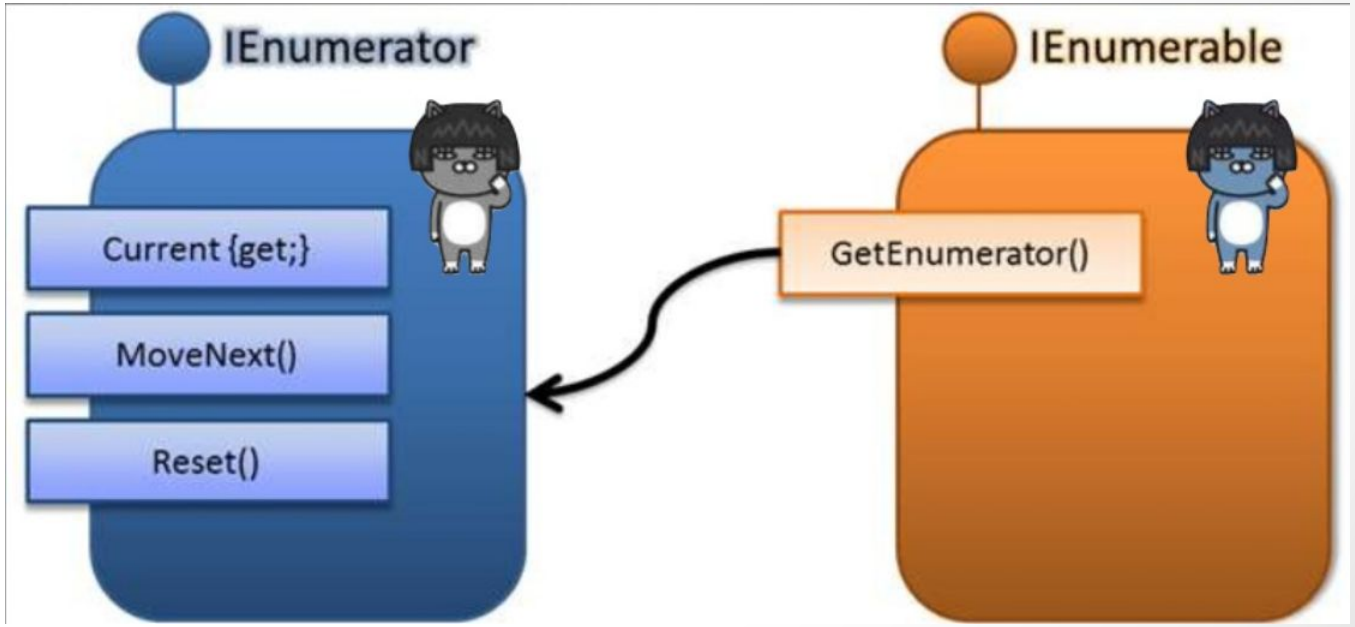
코루틴이란?

- IEnumerator를 생성해서 해결!
  - 넘겨줄때마다 IEnumerator를 한개씩 사용

N N+1 N+2 N+3 N+4

코루틴이란?

- IEnumerator는
  - 지금까지 시스템에서 몇번째까지 읽었는지(state)를 기억함
  - MoveNext를 호출받으면 다음 순번으로 이동해서 Current를 요구할때 해당 순번의 객체를 리턴함
  - 즉 순서대로 받아오고 싶은 족이 IEnumerable 에게서 IEnumrator 를 한개씩 받음



코루틴이란?

- 코루틴을 사용하면 IEnumerator가 자동 생성됨
- 리턴된 Ienumerator를 받아서 MoveNext() 실행하면 코드의 앞부분이 실행됨, 그후 yield return new WaitForSeconds가 호출 되면 Update()에서 해당 시간이 지났는지확인후 MoveNext() 호출 함!

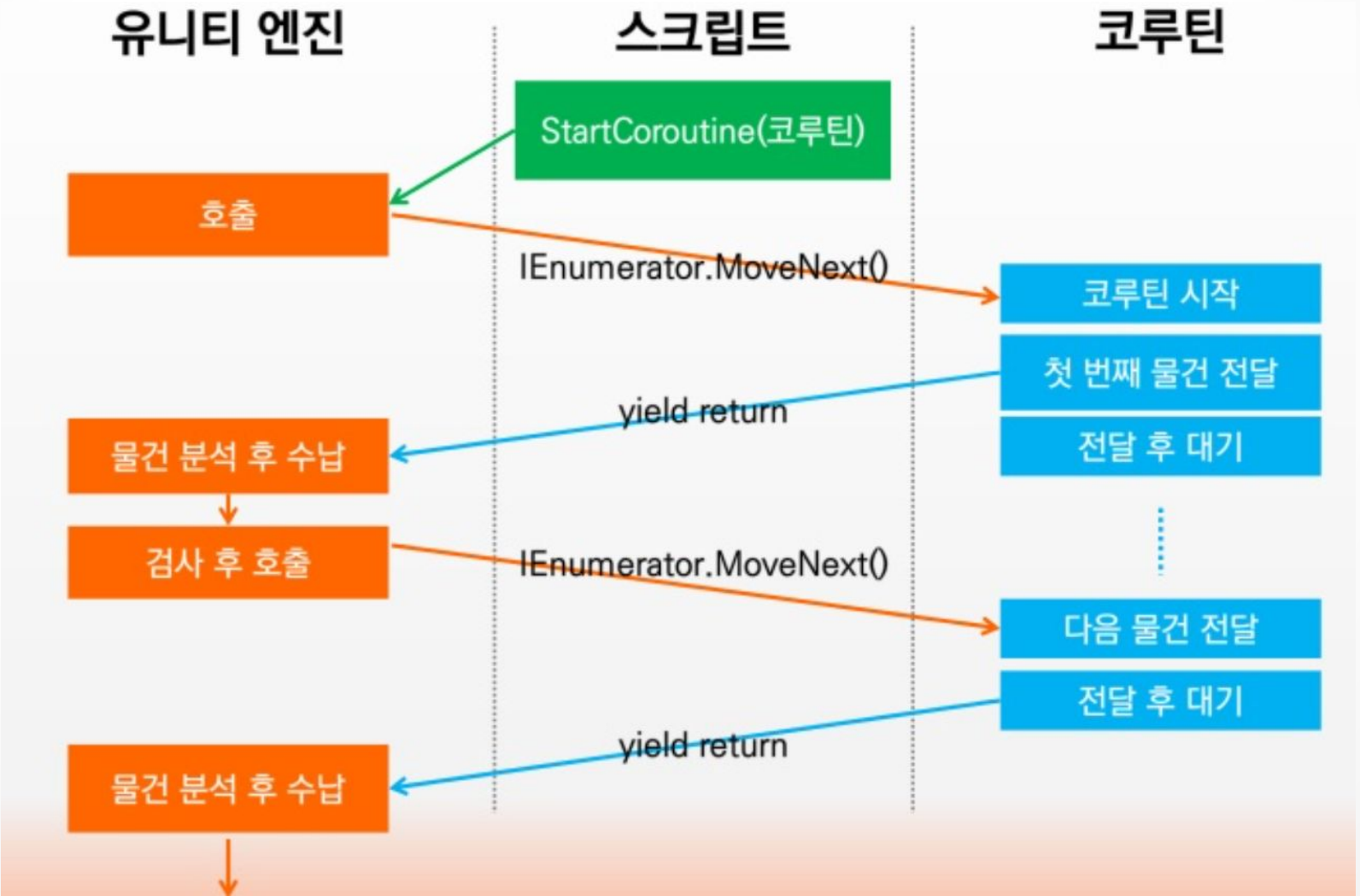
```
// 일정한 간격으로 몬스터의 행동 상태를 체크하고 monsterState의 값 변경
IEnumerator CheckMonsterState() {
    while(!isDie) {
        // 0.2초 동안 기다렸다가 다음으로 넘어감
        yield return new WaitForSeconds(0.2f);

        //몬스터와 플레이어 사이의 거리 측정
        float dist = Vector3.Distance(playerTr.position, monsterTr.position);

        if(dist <= attackDist){         // 공격거리 범위 이내로 들어왔는지 확인
            monsterState = MonsterState.attack;
        }
        else if(dist <= traceDist) {    //추적거리 범위 이내로 들어왔는지 확인
            monsterState = MonsterState.trace;  // 몬스터 상태를 추적으로 설정
        }
        else {
            monsterState = MonsterState.idle;   // 몬스터 상태를 idle모드로 설정
        }
    }
}
```

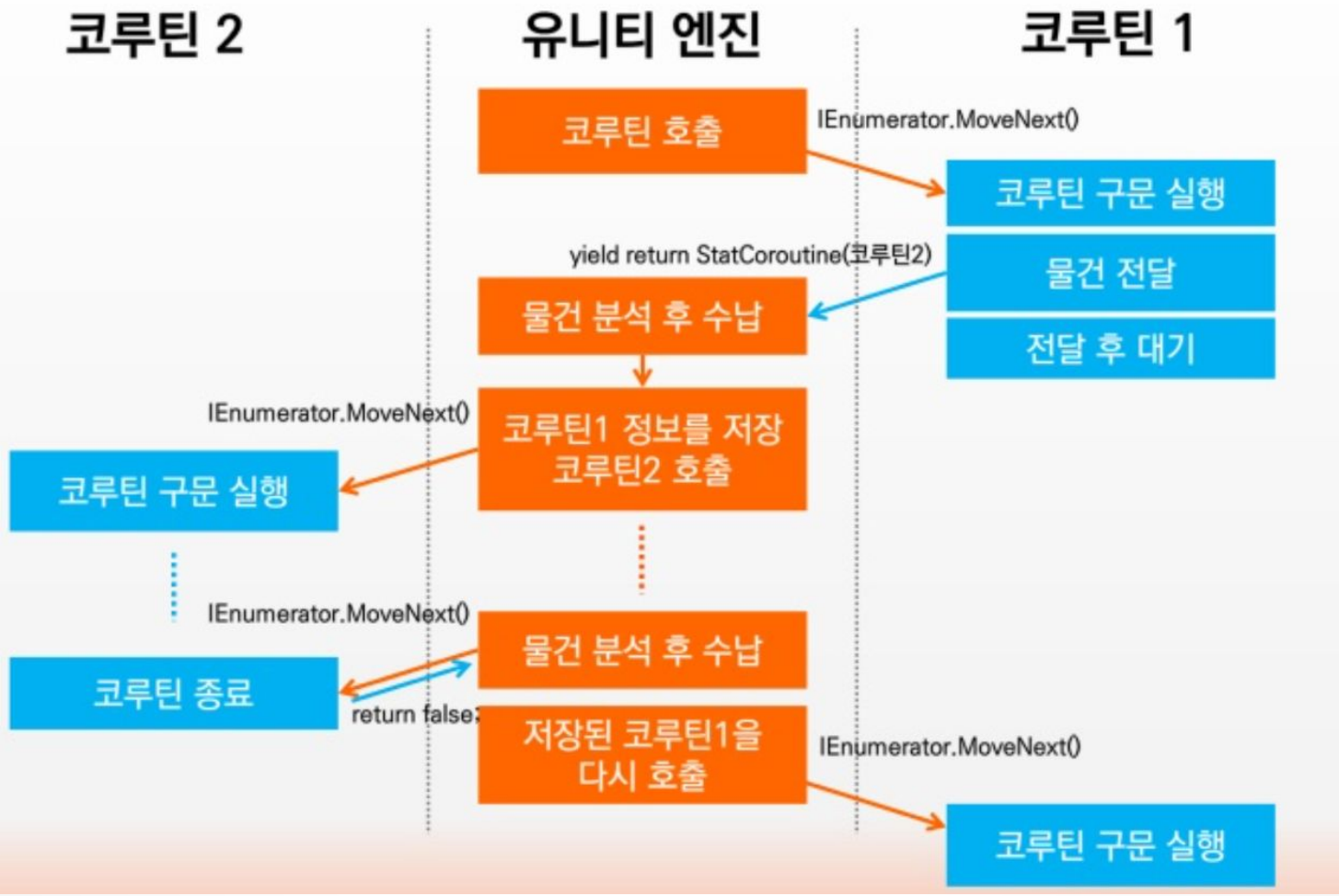
코루틴이란?

- 유니티 엔진과 코루틴의 정확한 관계 그림



코루틴이란?

- 코루틴은 정확히 말하면 멀티 쓰레드는 아님 동시 작업이 가능 하도록 구성된 콜백 함수 인터페이스에 가까움



코루틴 실습

- SpaceShooter 를 확인해 봅시다!
- 다음 링크에서 다운로드후 압축 풀기
- https://drive.google.com/file/d/1ytVTP_DtR3G6nBSCI90o2Iza83_XmIk0/view?usp=sharing

```
Start()

StartCoroutine(this.CheckMonsterState());
StartCoroutine(this.MonsterAction());
```

```
IEnumerator CheckMonsterState()
몬스터 상태 체크

if(플레이어와의 거리 <= 공격 사정거리)
    몬스터 상태 = attack
if(플레이어와의 거리 <= 추적 사정거리)
    몬스터 상태 = trace
else
    몬스터 상태 = idle

0.2초 동안 기다린 후 호출 (0.2초 간격)
```

```
IEnumerator MonsterAction()
몬스터 행동 수행

switch (몬스터 상태)
    case idle :
        trace 중지
        idle 애니메이션 수행
    case trace :
        공격 중지
        네비게이션 추적 재시작
        walk 애니메이션 수행
    case attack :
        추적 중지
        attack 애니메이션 수행

한 프레임 기다린 후에 호출
```

코루틴 실습

- MonsterCtrl.cs (1)

```
using UnityEngine;
using System.Collections;

public class MonsterCtrl : MonoBehaviour {

    // 몬스터 상태정보가 있는 Enumerable 변수 선언
    public enum MonsterState {idle, trace, attack, die};
    // 몬스터의 현재 상태 정보를 저장 할 Enum 변수
    public MonsterState monsterState = MonsterState.idle;

    // 속도 향상을 위해 각종 컴포넌트를 변수에 할당
    private Transform monsterTr;
    private Transform playerTr;
    private NavMeshAgent nvAgent;
    private Animator animator;

    public float traceDist = 10.0f;      // 추적 사정거리
    public float attackDist = 2.0f;      // 공격 사정거리
    private bool isDie = false;          // 몬스터 사망 여부
```

코루틴 실습

- MonsterCtrl.cs (2)

```
void Start () {
    monsterTr = this.gameObject.GetComponent<Transform>();                  // 몬스터의 Transform 할당
    playerTr = GameObject.FindWithTag("Player").GetComponent<Transform>();  // 추적대상인 Player의 Transform 할당
    nvAgent = this.gameObject.GetComponent<NavMeshAgent>();                 // NavMeshAgent 컴포넌트 할당
    animator = this.gameObject.GetComponent<Animator>();                    // Animator 컴포넌트 할당

    // 추적대상의 위치를 설정하면 바로 추적 시작
    // nvAgent.destination = playerTr.position;

    StartCoroutine(this.CheckMonsterState()); // 일정 간격으로 몬스터의 행동 상태를 체크하는 코루틴 함수 실행
    StartCoroutine(this.MonsterAction());     // 몬스터의 상태에 따라 동작하는 루틴을 실행하는 코루틴 함수 실행
}
```

코루틴 실습

- MonsterCtrl.cs (3)

```
// 일정한 간격으로 몬스터의 행동 상태를 체크하고 monsterState의 값 변경
IEnumerator CheckMonsterState() {
    while(!isDie) {
        // 0.2초 동안 기다렸다가 다음으로 넘어감
        yield return new WaitForSeconds(0.2f);

        //몬스터와 플레이어 사이의 거리 측정
        float dist = Vector3.Distance(playerTr.position, monsterTr.position);

        if(dist <= attackDist){          // 공격거리 범위 이내로 들어왔는지 확인
            monsterState = MonsterState.attack;
        }
        else if(dist <= traceDist) {     //추적거리 범위 이내로 들어왔는지 확인
            monsterState = MonsterState.trace;  // 몬스터 상태를 추적으로 설정
        }
        else {
            monsterState = MonsterState.idle;   // 몬스터 상태를 idle모드로 설정
        }
    }
}
```

코루틴 실습

- **MonsterCtrl.cs (4)**

```
// 몬스터의 상태값에 따라 적절한 동작을 수행하는 함수
IEnumerator MonsterAction() {
    while (!isDie) {
        switch(monsterState){
            // idle 상태
            case MonsterState.idle:
            nvAgent.isStopped = true;      / 추적중지
            animator.SetBool("IsTrace", false); // Animator의 IsTrace 변수를 false로 설정
            break;
            // 추적 상태
            case MonsterState.trace:
            nvAgent.destination = playerTr.position; // 추적 대상의 위치를 넘겨줌
            nvAgent.isStopped = false;     / 추적을 재시작
            animator.SetBool("IsTrace", true); // Animator의 IsTrace 변수값을 true로 설정
            break;
            // 공격 상태
            case MonsterState.attack:
            break;
        }
        yield return null;
    }
}
}
```

멀티 스레드를 써야하는 경우

- 코루틴의 문제점
  - 엄밀히 말마녀 멀티 스레드가 아니기 때문에 싱글 스레드만 사용함
  - 자원의 낭비가 심함… 대부분의 게임들은 멀티 스레드를 풀로 활용해야 속도를 맞출수 있음

- 멀티스레드를 게임 프로그래밍에 사용하는 요소들
  - 시뮬레이션 (길찾기, 건설 등의 작업들)
  - AI 작업들 (AI를 수행할 몬스터 및 NPC가 매우 많은 경우)
  - 그리고 네트워크를 통해서 서버와 통신하는 경우
    - 서버는 멀티 스레드 프로그래밍 & 스케쥴링이 필수임
  - 대부분 스레드 풀 & Queue를 활용한 스케쥴링구조를 가짐

멀티 스레드 실습해보기

- 유니티에서 멀티 스레드 사용시 주의점
  - UI 나 애니메이션 부분 접근은 하지 못함 -> Update 에서 할 수 있도록 해야함
  - Lock을 거는 것도 마찬가지로 사용하지 못함

멀티 스레드 실습해보기

- MonsterCtrl.cs 파일을 다음과 같이 수정하기

```
using UnityEngine.AI;
using System.Collections;
using System.Threading;

public class MonsterCtrl : MonoBehaviour
{
    public enum MonsterState { idle, trace, attack, die};
    public MonsterState monsterState = MonsterState.idle;
    private Transform monsterTr;
    private Transform playerTr;
    private NavMeshAgent nvAgent;
    private Animator animator;

    public float traceDist = 10.0f;
    public float attackDist = 2.0f;
    private bool isDie = false;

    private int hp = 100;

    private Thread thread = null;
    private float distance = 0f;
```

멀티 스레드 실습해보기

- MonsterCtrl.cs 파일을 다음과 같이 수정하기

```
50      void CheckMonsterThread()
51      {
52          while (isDie == false)
53          {
54              Thread.Sleep(200);
55
56              if (monsterState == MonsterState.die)
57              {
58                  ;
59              }
60              else if (distance <= attackDist)
61              {
62                  monsterState = MonsterState.attack;
63                  Debug.Log("CheckMonsterState(): attack!");
64              }
65              else if (distance <= traceDist)
66              {
67                  monsterState = MonsterState.trace;
68                  Debug.Log("CheckMonsterState(): trace!");
69              }
70              else
71              {
72                  monsterState = MonsterState.idle;
73                  Debug.Log("CheckMonsterState(): idle!");
74              }
75          }
76      }
77
```

멀티 스레드 실습해보기

- MonsterCtrl.cs 파일을 다음과 같이 수정하기

```
// Update is called once per frame
void Update()
{
    distance = Vector3.Distance(playerTr.position, monsterTr.position);
}
```

```
// Start is called before the first frame update
void Start()
{
    monsterTr = GetComponent<Transform>();
    playerTr = GameObject.FindWithTag("Player").GetComponent<Transform>();
    nvAgent = GetComponent<NavMeshAgent>();
    animator = GetComponent<Animator>();
    //nvAgent.destination = playerTr.position;

    //StartCoroutine(this.CheckMonsterState());
    StartCoroutine(this.MonsterAction());

    thread = new Thread(CheckMonsterThread);
    thread.Start();
}

private void OnDestroy()
{
    thread.Abort();
    thread = null;
}
```

멀티 스레드 실습해보기

- 코루틴 MonsterAction은 고려사항이 많음!
  - 유니티 UI 관련 함수들을 너무 많이 사용함!
  - 멀티스레드로 바꾸려면 다음과 같이 처리해야함!

```
    // 몬스터의 상태값에 따라 적절한 동작을 수행하는 함수
    IEnumerator MonsterAction() {
        while (!isDie) {
            switch(monsterState){
                // idle 상태
                case MonsterState.idle:
                nvAgent.Stop();     // 추적중지
                animator.SetBool("IsTrace", false); // Animator의 IsTrace 변수를 false로 설정
                break;
                // 추적 상태
                case MonsterState.trace:
                nvAgent.destination = playerTr.position; // 추적 대상의 위치를 넘겨줌
                nvAgent.Resume();   // 추적을 재시작
                animator.SetBool("IsTrace", true); // Animator의 IsTrace 변수값을 true로 설정
                break;
                // 공격 상태
                case MonsterState.attack:
                break;
            }
            yield return null;
        }
    }
}
```

멀티 스레드 실습해보기

- Queue 추가!

MonsterCtrl.cs

Assembly-CSharp

MonsterCtrl

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using System.Collections;
using System.Threading;

public class MonsterCtrl : MonoBehaviour
{
    public enum MonsterState { idle, trace, attack, die};
    public MonsterState monsterState = MonsterState.idle;
    private Transform monsterTr;
    private Transform playerTr;
    private NavMeshAgent nvAgent;
    private Animator animator;

    public float traceDist = 10.0f;
    public float attackDist = 2.0f;
    private bool isDie = false;

    private int hp = 100;

    private Thread thread = null;
    private float distance = 0f;
    private object lockObject = new object();
    private static Queue<MonsterState> TaskQueue = new Queue<MonsterState>();
```

멀티 스레드 실습해보기

- Start 함수에 코루틴들은 다 주석 처리!

```
// Start is called before the first frame update
void Start()
{
    monsterTr = GetComponent<Transform>();
    playerTr = GameObject.FindWithTag("Player").GetComponent<Transform>();
    nvAgent = GetComponent<NavMeshAgent>();
    animator = GetComponent<Animator>();
    //nvAgent.destination = playerTr.position;

    //StartCoroutine(this.CheckMonsterState());
    //StartCoroutine(this.MonsterAction());
    thread = new Thread(CheckMonsterThread);
    thread.Start();



}

private void OnDestroy()
```

멀티 스레드 실

- Lock을 걸면 성능이 느려짐.. Queue를 사용하는 부분에만 걸기..
  - 사실 lock을 안쓸수 있으면 제일 좋음

```
{
    while (isDie == false)
    {
        Thread.Sleep(200);

        if (monsterState == MonsterState.die)
        {
            ;
        }
        else if (distance <= attackDist)
        {
            lock (lockObject)
            {
                TaskQueue.Enqueue(MonsterState.attack);
            }
            Debug.Log("CheckMonsterState(): attack!");
        }
        else if (distance <= traceDist)
        {
            lock (lockObject)
            {
                TaskQueue.Enqueue(MonsterState.trace);
            }
            Debug.Log("CheckMonsterState(): trace!");
        }
        else
        {
            lock (lockObject)
            {
                TaskQueue.Enqueue(MonsterState.idle);
            }
            Debug.Log("CheckMonsterState(): idle!");
        }
    }
}
```

멀티 스레드

- Update 함수 업데이트
  - 유니티 핵심 처리를 이쪽해서 해야함!

```
// Update is called once per frame
void Update()
{
    distance = Vector3.Distance(playerTr.position, monsterTr.position);

    if(TaskQueue.Count > 0)
    {
        lock (lockObject)
        {
            monsterState = TaskQueue.Dequeue();
        }
        switch (monsterState)
        {
            case MonsterState.idle:
                nvAgent.isStopped = true;
                animator.SetBool("IsTrace", false);

                break;
            case MonsterState.trace:
                nvAgent.destination = playerTr.position;
                nvAgent.isStopped = false;
                animator.SetBool("IsAttack", false);
                animator.SetBool("IsTrace", true);

                break;
            case MonsterState.attack:
                nvAgent.isStopped = true;
                animator.SetBool("IsAttack", true);
                break;
        }
    }
}
```

멀티 스레드 실습해보기

- 유니티에서 멀티 쓰레드 사용 고려할점!
  - 유니티의 기본 기능 사용이 어려움 – 해당 값들은 Update 등으로 옮겨야함
  - 쓰레드가 여러개 일때는 Queue를 사용하여 처리하도록 구성해야함
  - 네트워크 모듈을 제외한 게임 모듈 중 쓰레드를 쓰기 좋은 모듈에만 잘 적용해야함
    - 천 이상의 객체들이 개별적으로 움직이면서 시뮬레이션 (건설, 놀이기구타기, 특정 작업 및 길찾기 등의 AI등을 하는 경우)… 이 경우에는 쓰레드를 코어 개수 만큼 만들어서 운영