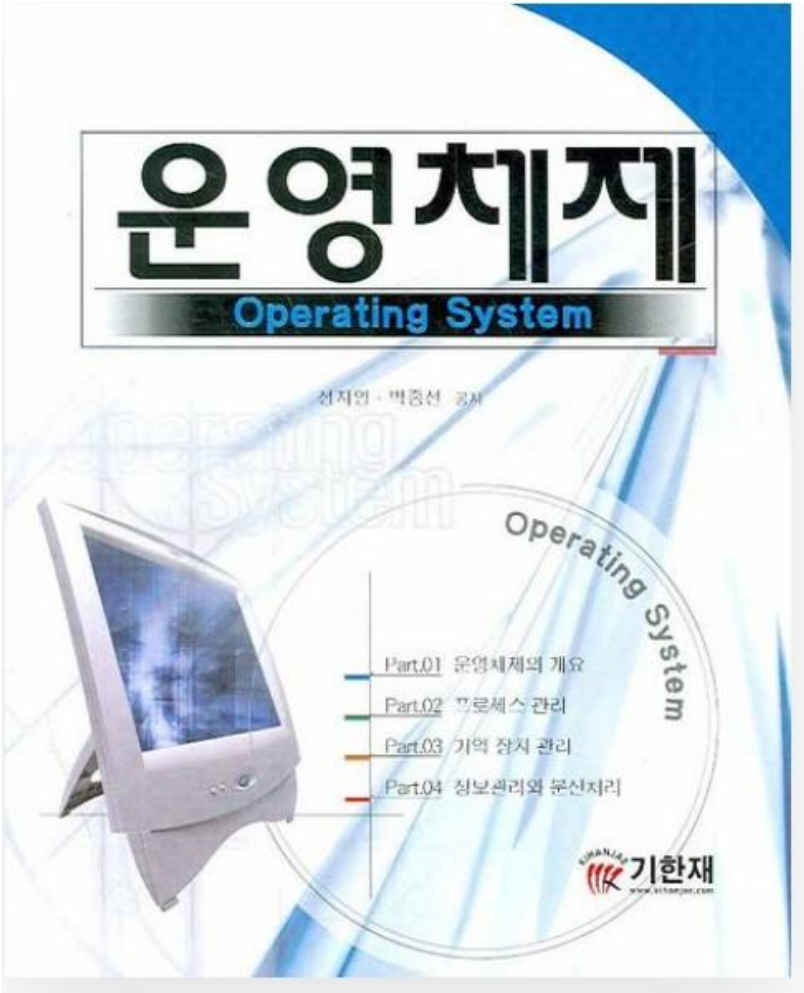
R-eve

지도교수 : 정지영

https://url.kr/fceigz

병렬 Process

운영체제 2010.10 정지영, 박종선 공저, 절판



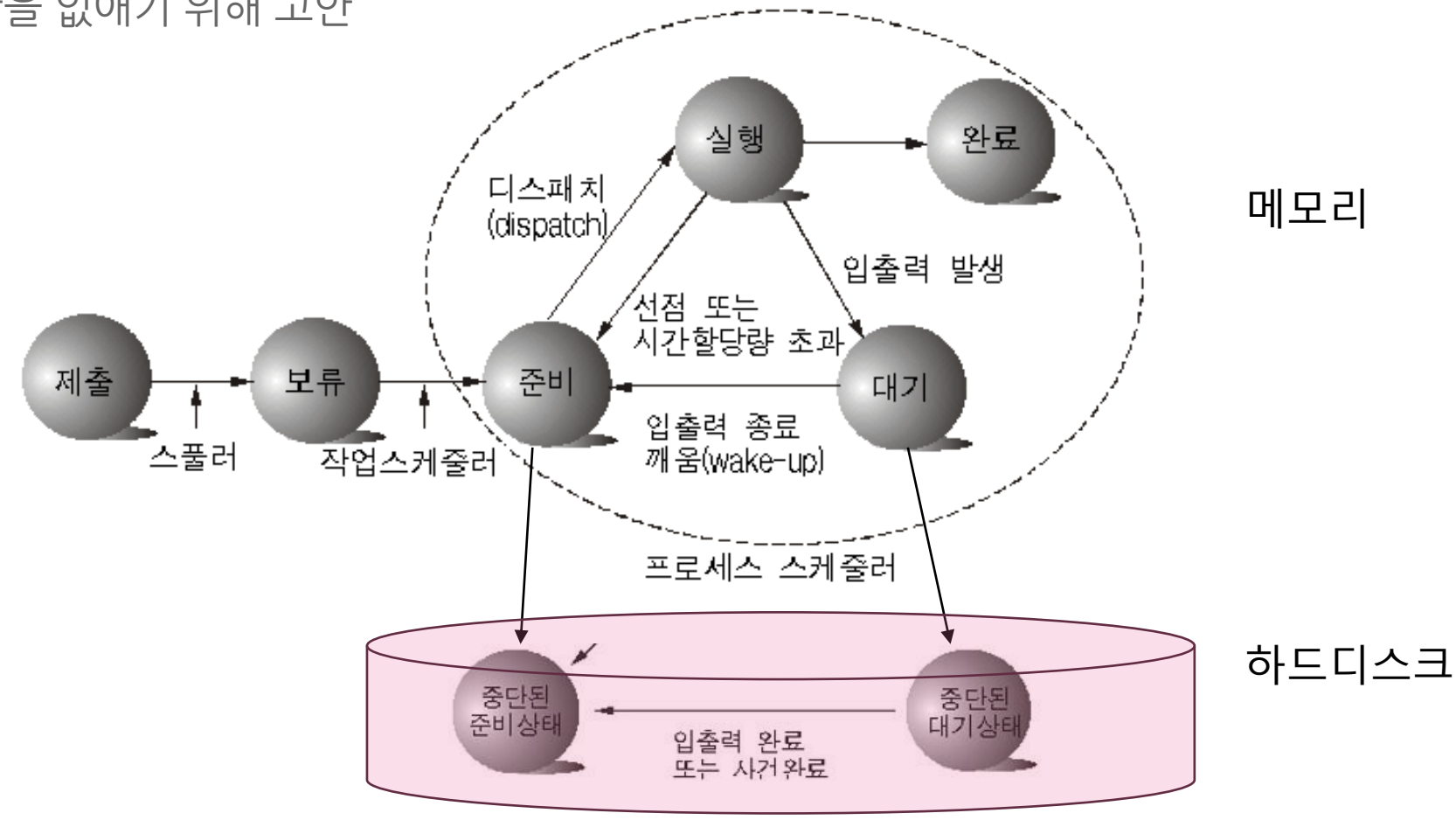
ISBN 9788970184098

프로세스(Process)의 정의

- **CPU에 의해 수행되는 프로그램**
- 컴퓨터 시스템은 시스템 프로세스와 사용자 프로세스들의 집합체
- 작업의 기본 단위로 시스템 내의 모든 활동 요소
- **현재 실행중인** 또는 실행 가능한 프로그램, 또는 살아 있는 프로그램
- 운영체제 내에 PCB(Process Control Block)를 가진 프로그램
- 프로세스가 할당하는 개체로서 디스패치가 가능한 단위
- dispatch : OS가 다음에 실행할 프로그램을 결정해 CPU 사용권을 주는 조작

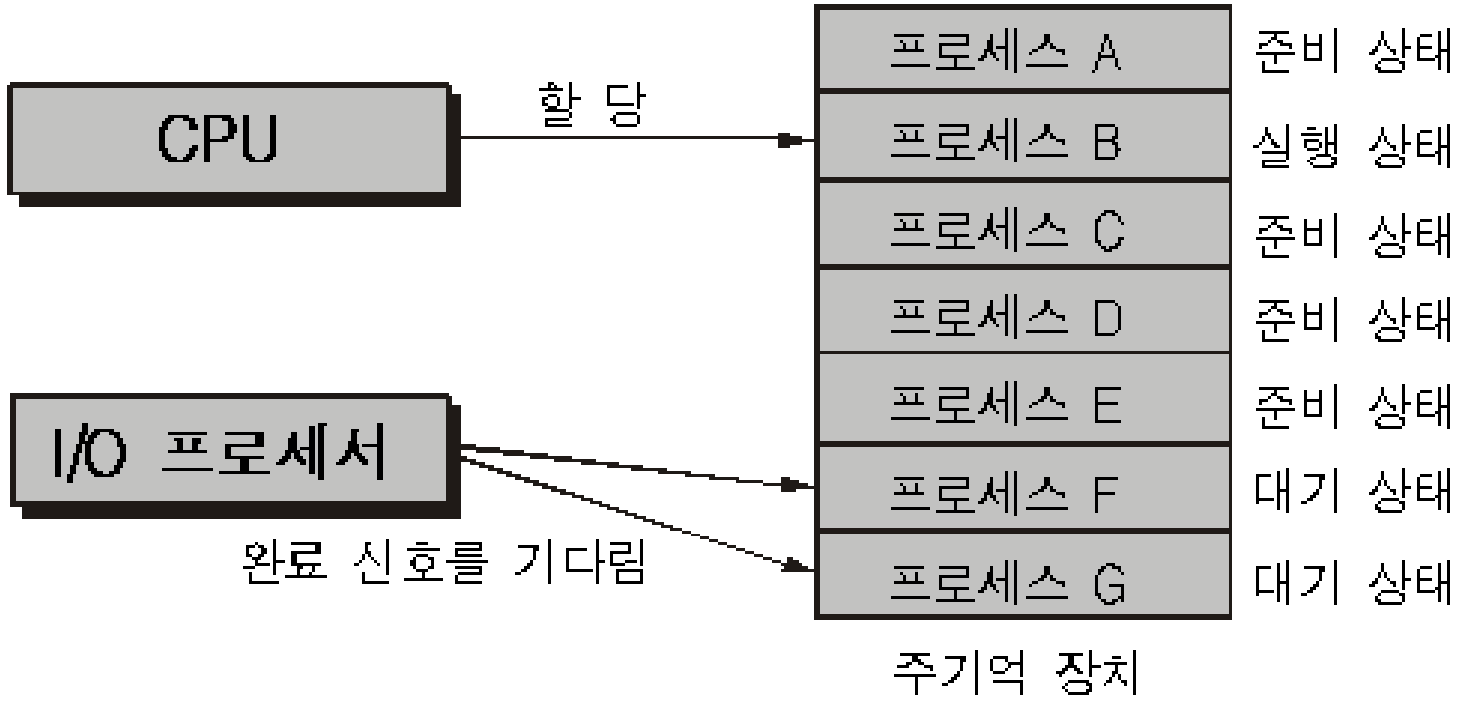
프로세스 상태

스풀(SPOOL) : 컴퓨터 하드디스크 드라이브의 임시저장 방식, 주변장치와 컴퓨터의 처리속도가 달라 발생하는 대기시간을 없애기 위해 고안



프로세스 상태

context switching : 한 프로세스에서 다른 프로세스로 CPU가 할당되는 과정
time slice : 각각의 프로세스에게 지정된 시간 동안 CPU를 점유할 수 있도록 한 시간



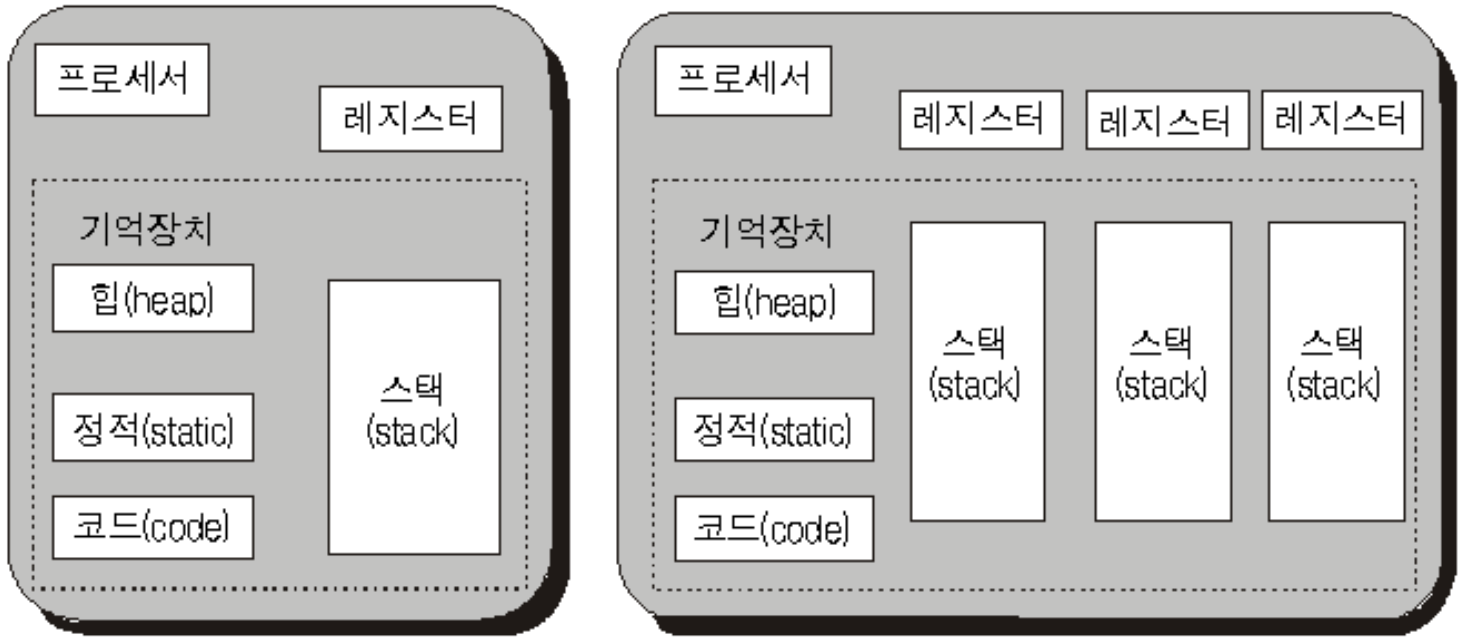
스레드(Thread)

- **스레드(Thread) = Light Process**
  - 프로세스나 태스크(task)보다 더 작은 단위
  - 다중 프로그래밍 시스템에서 CPU에게 보내져 실행되는 프로그램 단위
  - 프로세스 스케줄링 부담을 줄여 성능을 향상시키기 위한 프로세스의 다른 표현 방식

- **스레드의 개념**
  - 단일 프로세스용 운영체제에서 프로세스를 생성하고 유지하는데 많은 비용을 필요로 하는 문제와 병렬 처리가 불가능한 문제를 개선하기 위한 방법으로 **스레드를 기본 단위로 하는 다중 스레딩이란 개념이 도입**
  - **스레드란** 제어의 흐름을 의미하는 것으로 프로세스의 구성을 크게 제어의 흐름 부분과 실행 환경 부분으로 나눌 때 스레드는 프로세스의 실행 부분을 담당함으로써 **실행의 기본 단위**가 된다.

단일 & 다중 스레드형 프로세스

- 다중 스레드형 프로세스는 **자기 고유의 레지스터와 스택을 가짐**



(a) 단일 스레드형 프로세스

(b) 다중 스레드형 프로세스

static, heap, stack

- **Static 영역**

  클래스에 고정: 프로그램이 종료될 때까지 메모리에 존재

- **Stack 영역**

  기본 자료형을 생성할 때 저장하는 공간: 지역 변수, 매개변수가 저장되는 영역

  메서드를 호출할 때마다 스택이 생성되며 메모리에 할당되고 종료시 영역에서 해제

  heap 영역의 주소 값을 stack 영역의 객체가 가짐

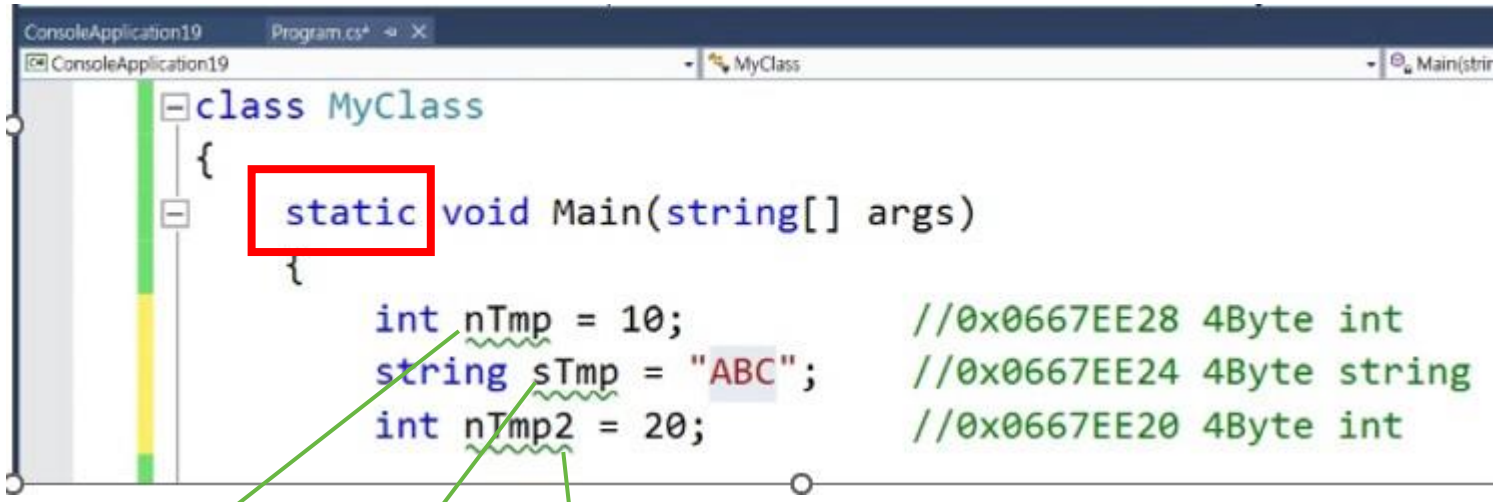
- **Heap 영역**

  참조형 데이터 타입의(new 키워드로 생성) 객체는 Heap 영역에 저장

  동적 할당 영역(인스턴스의 실제 데이터는 Heap영역에 올라감)

  프로그램이 실행되는 도중에 내가 원할 때 띄울 수 있는 영역

변수의 메모리 할당


ConsoleApplication19
Program.cs*
ConsoleApplication19
MyClass
Main(strir
class MyClass
{
static void Main(string[] args)
{
int nTmp = 10; //0x0667EE28 4Byte int
string sTmp = "ABC"; //0x0667EE24 4Byte string
int nTmp2 = 20; //0x0667EE20 4Byte int


nTmp
sTmp
nTmp2
nTmp3
10
3FE7D7
10
10
value Type 변수
레퍼런스타입변수
value Type 변수
지역변수 + 매개변수
stack
cds cds cds cds cds A B C E F G
heap
동적할당
데이터의 크기가 변형되는 스트링,
sTmp = sTmp + "EFG"
data
전역변수 + static
영역 (코드/전역)
객체를 선언만 해도 메모리가 할당

스레드와 프로세스 차이

하나의 프로세스에 다수의 실행 단위들이 존재하여 작업 수행에 필요한 자원들을 공유하기 때문에 자원의 생성 및 관리가 중복되는 것을 최소화

구분 / 항목	스레드	프로세스
실행점	CPU가 하나인 시스템에서 실행 중인 프로세스에는 여러 개의 스레드가 존재하므로 실행점이 여러 개임	CPU가 하나인 시스템에서는 하나의 프로세스만 실행되므로 실행점이 하나임
병행성	CPU가 하나인 시스템에서도 병행 실행이 가능함	CPU가 하나인 시스템에서는 순차적으로 실행됨

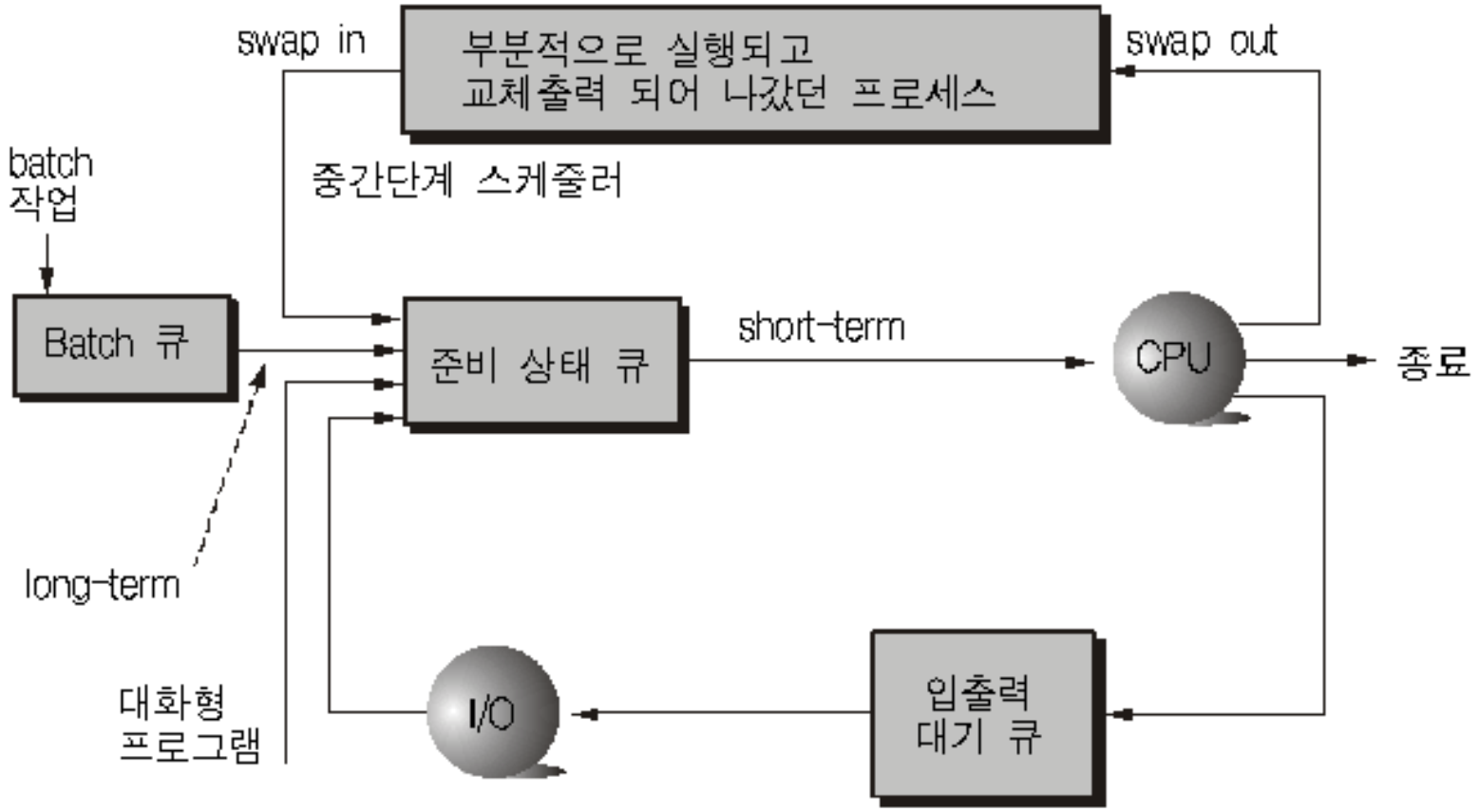
스케줄링 알고리즘

스케줄링 성능 기준

기 준
CPU 이용률(utilization)
처리율(throughput)
반환 시간(turn around time)
대기 시간(waiting time)
응답 시간(response time)

CPU 스케줄링	기능별 분류	▪ 작업(job) 스케줄링
		▪ CPU(processor) 스케줄링
	방법별 분류	▪ 선점(preemptive) 스케줄링
		▪ 비선점(non-preemptive) 스케줄링
	알고리즘별 분류	▪ 우선순위(priority) 스케줄링
		▪ 기한부(deadline) 스케줄링
		▪ FCFS(First Come First Sever) 스케줄링
		▪ RR(Round Robin) 스케줄링
		▪ SJF(shortest job first) 스케줄링
		▪ SRT(shortest remaining time) 스케줄링
		▪ HRN(highest response ratio next) 스케줄링
		▪ MLQ(multi level queue) 스케줄링
		▪ MFQ(multilevel feedback queue) 스케줄링

스케줄러의 큐잉

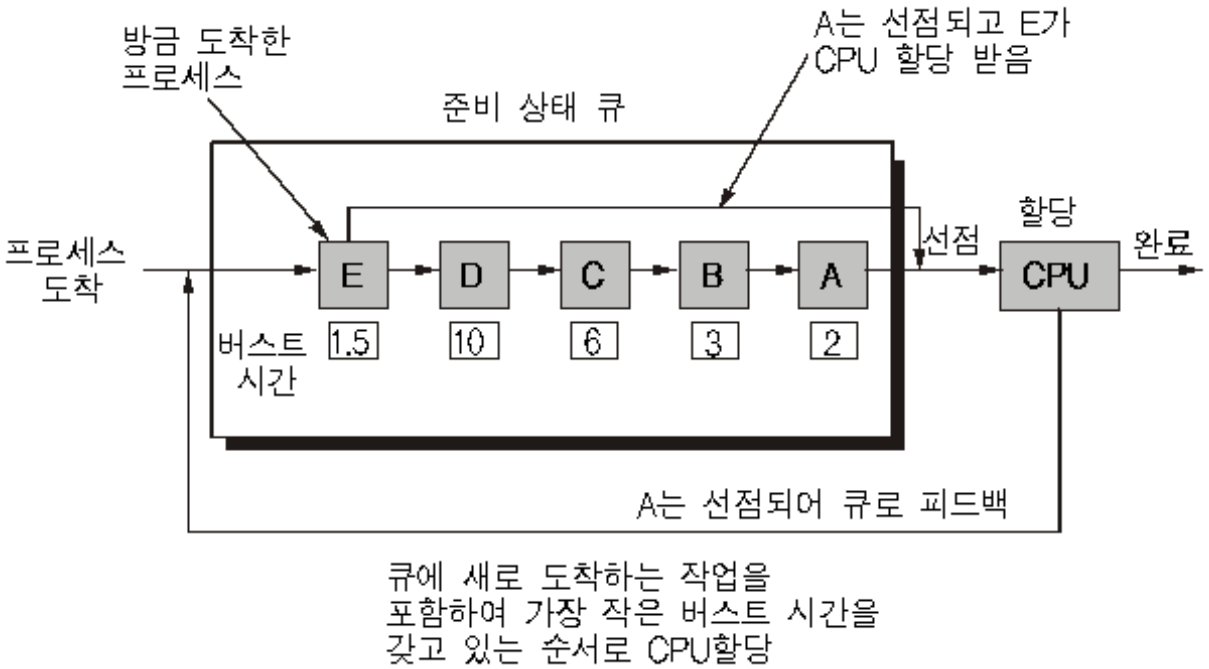

swap in
부분적으로 실행되고
교체출력 되어 나갔던 프로세스
swap out
batch
작업
중간단계 스케줄러
Batch 큐
준비 상태 큐
short-term
CPU
종료
long-term
대화형
프로그램
I/O
입출력
대기 큐

스케줄링의 기능별 분류

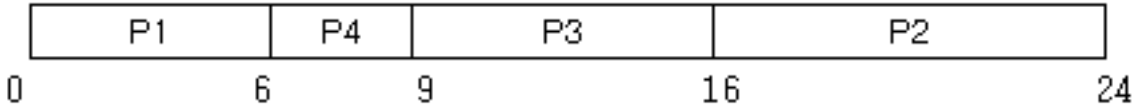
작업 스케줄링	**· 상위 수준(long term) 스케줄링**, 장기 스케줄러에 의해 수행 **· 어떤 작업이 시스템의 자원들을 차지할 수 있도록 할 것인가를 결정** **· 프로세스가 시스템을 떠나는 경우에 가끔씩 실행**되며, **호출 빈도수가 적음**
CPU 스케줄링	**· 하위 수준(short term) 스케줄링**, 단기 스케줄러에 의해 수행 **· 디스패처(dispatcher)에 의해서 매초 여러 번 수행** **· 다중 프로그래밍과 시분할 작업 등을 위해 자주 수행, 수행 속도가 빠름**
중기 스케줄링	**· 중위 수준(medium-term) 스케줄링**, 중기 스케줄러에 의해 수행 **· 어떤 프로세스들이 CPU를 차지할 것인가를 결정** **· 시스템의 단기적인 부하를 조절하는 버퍼 역할** · 일시 보류된 프로세스는 디스크에 저장되며, 후에 주기억장치로 적재

프로세스(단기) 스케줄링 예

- **SRT 스케줄링**



프로세스	도착시간	버스트시간	대기 시간	완환 시간
P1	0	6	0	6
P2	1	8	15	23
P3	2	7	7	14
P4	3	3	3	6



평균 대기 시간 : (0+15+7+3)/4 = 6.25ms
평균 반환 시간 : (6+23+14+6)/4 = 12.25ms

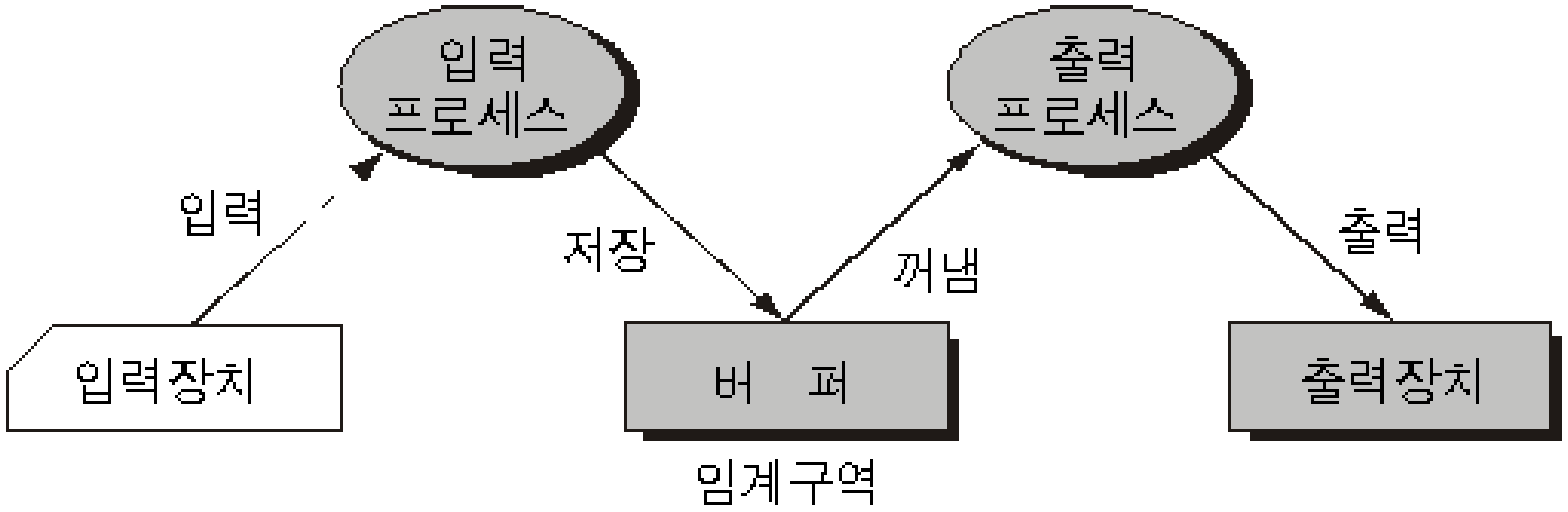
병행 프로세스(concurrent process)

- **여러 프로세스가 병행 처리될 때의 결정성(determinancy) 보장**
  - 공유 자원의 상호 배제(mutual exclusion)문제
  - 프로세스의 교착 상태(deadlock)문제
  - 하나의 기능을 함께 수행하는 프로세스간의 동기화(synchronization) 문제

  - determinancy : 실행순서와 관계없이 항상 같은 결과를 얻는것

상호배제 및 임계구역

- **상호배제** : 두개 이상의 프로세스들이 동시에 임계구역에 있어서는 안됨
- **진행** : 임계구역 바깥에 있는 프로세스가 다른 프로세스의 임계구역 진입을 막으면 안됨
- **한계 대기** : 어떤 프로세스도 임계 구역으로 들어가는 것이 무한정 연기되어서는 안됨



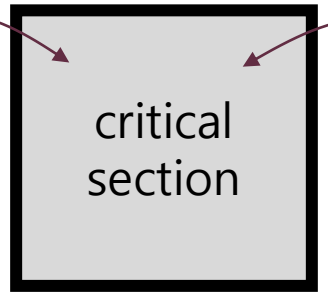
동시에 버퍼를 사용하면 안됨

상호 배제 [알고리즘 1]

```
while (1) {                                  i=0, j = 1;
        . . .
        while (turn != i);
        // 임계 구역(critical section)
         . . .
        turn =  j;
        // 잔류 구역(remainder section)
         . . .
    }
```

공유변수

turn =0;




```
while (1) {                                  i=0, j = 1;
        . . .
        while (turn != j);
        // 임계 구역(critical section)
         . . .
        turn = i;
        // 잔류 구역(remainder section)
         . . .
    }
```

?

상호 배제 [알고리즘 2]

```
[while (1) {                                    0
        . . .
        flag[0] = true;
        while (flag[1] == true);
        // 임계 구역(critical section)
        . . .
         flag[0] = false;
        // 잔류 구역(remainder section)
         . . .
  }
```

```
flag[0] = false
flag[1] = false
```

critical section

```
while (1) {                                     1
        . . .
        flag[1] = true;
        while (flag[0] == true);
        // 임계 구역(critical section)
        . . .
         flag[1] = false;
        // 잔류 구역(remainder section)
         . . .
  }
```

?

Lamport의 빵집 알고리즘

분산 처리 시스템에 유용한 알고리즘, 고객이 많은 빵집에서 번호표를 받고 대기하는 개념 사용

```
while (1) {
   . . .
   choosing[i] : = true; // 번호표를 뽑기 전
   number[i] = max(number[0], number[1],.....number[n-1] + 1;
   choosing[i] = false; // 번호표를 뽑은 후
   for (j : =0; j < n; J ++) { //모든 프로세스에 대한 번호표 비교루프
     while (choosing[j]); // Pj가 번호표 받을 때까지 대기
     while (number[j] != 0 && (number[j], j) < (number [i], i));
   }
   //임계 구역(critical section)
   . . .
   number[i : ] = 0;
   // 잔류 구역(remainder section)
   . . .
}
```

세마포어(semaphore)

정수형 변수로 초기화 및 두 개의 연산 P와 V로만 접근 가능한 특수한 변수

P(s)

```
if (s.value > 0)
        s. value--;
else
        현재의 프로세스를 s. list에서 기다린다.
```

V(s)

```
if (1개 이상의 프로세스가 s에서 대기 중이면)
        그 중 1개 프로세스만 진행
else
        s. value++;
```

s =1

```
while (1) {
        ⋮
        P(s);
        임계 구역(critical section)
        ⋮
        V(s);
        잔류 구역(remainder section)
        ⋮
}
```

스레드(Thread) 실습

- **응용예제 11-1 스레드** 인스턴스 생성 : 생성자 매개변수로 [메서드 이름, 무명 델리게이트, 람다]
- ① 스레드 생성하기

코드 11-10 Thread 클래스의 인스턴스 생성 동시에 코드실행: 내부적으로는 번갈아 가며 실행

```
class Program
{
    static void Main(string[] args)
    {
        Thread threadA = new Thread(TestMethod);
        Thread threadB = new Thread(delegate()
        {

        });
        Thread threadC = new Thread(() => {

        });
    }
    public static void TestMethod()
    {

    }
}
```

메서드 이름

무명 델리게이트

람다

스레드(Thread) 실습

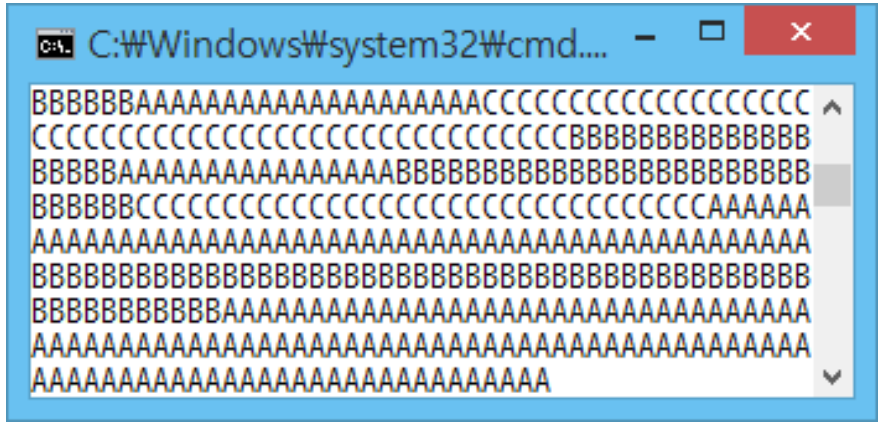
② 스레드 실행하기

코드 11-11 스레드 실행

```
static void Main(string[] args)
{
    Thread threadA = new Thread(() => {
        for(int i = 0; i < 1000; i++) {
            Console.Write("A") ;
        }
    });
    Thread threadB = new Thread(() => {
        for(int i = 0; i < 1000; i++) {
            Console.Write("B");
        }
    });
    Thread threadC = new Thread(() =>
    {
        for (int i = 0; i < 1000; i++)
        {
            Console.Write("C");
        }
    });

    threadA.Start();
    threadB.Start();
    threadC.Start();
}
```

Start() 메서드를 호출하여 해당 쓰레드 작업에 필요한 메모리를 할당받고 주어진 작업 시작



Task : .NET 4.0에 도입된 클래스로서 쓰레드풀로부터 쓰레드를 가져와 비동기 작업을 실행

- 스레드는 계속 생성/삭제하면 오버헤드 발생
- Task는 기본적으로 백그라운드 속성의 스레드
- 반환 값이 없는 Task(Action), 있는 Task(Func<T>)
- 결과값을 리턴 받을 수 있음
  - Int result = taskName.**Result**; // 리턴값 저장
- Task.Run() : static 메소드로 스레드 실행 가능
- 종료 할 때까지 대기하는 메소드도 존재
  - taskName.**Wait**();
- using System.Threading.Tasks;

```
static void Main(string[] args)
{
    // 반환값 없는 Task(Action)
    Task t1 = new Task(() =>  // A 1000...);
    Task t2 = new Task(Mytask); // B 1000

    // 반환값 있는 Task(Func<T>)
    Task<int> t3 = new Task<int>(TaskFunc, "Hello ");

    t1.Start(); t2.Start(); t3.Start();
    Console.WriteLine(t3.Result);
    Task.Run(Mytask);

    t1.Wait(); t2.Wait(); t3.Wait(); // 실행완료 대기
    // or Console.ReadKey();
}

// Task.Run() : static 메소드로 스레드 실행 가능
private static int TaskFunc(object obj)
{
    Console.Write(" ::: " + obj);
    return 1000;
}

static void Mytask()
{
    for (int i = 0; i < 1000; i++)
    {
        Console.Write("B");
    }
}
```

- 작업들의 처리 순서가 있을 경우 async/await 사용 (C# 5.0 이상)
  - ContinueWith()를 이용해 처리가능하나 코드 작성이 힘들며 코드가 분리 되야 하는 단점
- 비동기 메소드 정의에 **async** 키워드 사용
- 대기할 작업 앞에 **await** 키워드를 사용
- Task<T>의 경우 await을 적용하면 내부적으로

  T타입으로 변경 후 리턴

```
private static async void SumofCalc() {
    // 다른 작업 호출
    Task<long> task = Task.Run(other);
    // 자신의 작업 수행
    long fsum =  await task; // 다른 작업 완료 대기
```

C# : Semaphore

```
private static Semaphore semaphore = new Semaphore(1,1);
static void Main(string[] args)  {
        ......
        ChasingGhosts();
}
private static void ChasingGhosts()
        ......
        semaphore.WaitOne();    //-----------------
        Console.SetCursorPosition(gx, gy);
        Console.Write(" ");
        gx += mvx; gy += mvy;
        Console.SetCursorPosition(gx, gy);
        Console.Write("G");
        semaphore.Release();  //-----------------
```

```
Semaphore(int initialCount, int maximumCount);
```

✓ SemaphoreSlim 클래스는 대기 시간이 짧은 프로세스 내에서 대기하는데 사용하는 가볍고 빠른 세마포어

C# 버전

컴파일러는 다음 규칙에 따라 기본값을 결정합니다.

Target	버전	C# 언어 버전 기본값
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	모두	C# 7.3

Unity의 코루틴(Coroutine)

- 코루틴은 Multi-Thread와 유사한 개념의 병렬처리 루틴
- **유니티 공식 메뉴얼** : 코루틴은 실행을 중지하여 Unity에 제어권을 돌려주고, 다음 프레임에서 중지한 곳부터 실행을 계속할 수 있는 기능.
- 코루틴은 yield return을 통해 현재 위치를 기억하고 다음 반복에서 다음 요소를 반환하는 IEnumerator(데이터를 하나씩 넘겨줄 때 사용되는 인터페이스)를 사용
- 유니티는 Thread-Safety를 지원하지 않으므로 비동기와 유사한 작업을 위해 코루틴을 사용
- 많은 수의 객체를 한번에 생성하거나 시간을 많이 소모하는 작업에 사용
- 싱글쓰레드로 돌기 때문에 멀티쓰레드의 자원관리, Context switching 등을 고려않해도 됨

✓ 코루틴 내부에서 다른 코루틴을 실행하여 반환하면 해당 코루틴이 끝날 때까지 대기

Coroutine 사용 예

```
StartCoroutine( CreateBullet() );

IEnumerator CreateBullet(){
        Instantiate ( bullet, firePos.position, firePos.rotation);
        yield return null;
        // yield return new WaitForSeconds(time );
}
```

코루틴용 데이터	엔진이 수행하는 기능
yield return null	다음 프레임까지 대기
yield return new WaitForSeconds(float)	지정된 초 만큼 대기
yield return new WaitForFixedUpdate()	다음 물리 프레임까지 대기
yield return new WaitForEndOfFrame()	모든 렌더링작업이 끝날 때까지 대기
yield return StartCoRoutine(string)	다른 코루틴이 끝날 때까지 대기
yield return new WWW(string)	웹 통신 작업이 끝날 때까지 대기
yield return new AsyncOperation	비동기 작업이 끝날 때까지 대기

- ✓ StartCoroutine, StopCoroutine, StopAllCoroutine
- ✓ yield : 넘겨주다

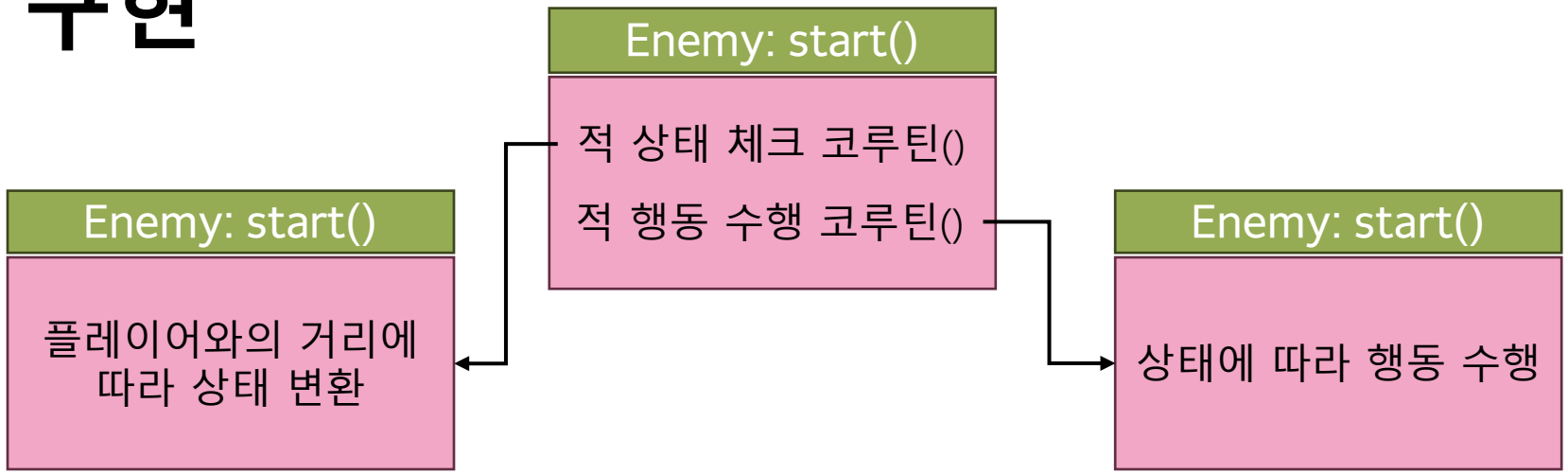
C# IEnumerator(열거자) [F11]로 확인

1. 최소 하나의 yield 문 필요
2. 변수처럼 대입하여 사용가능
3. MoveNext 메소드로 열거자 내부의 내용 실행

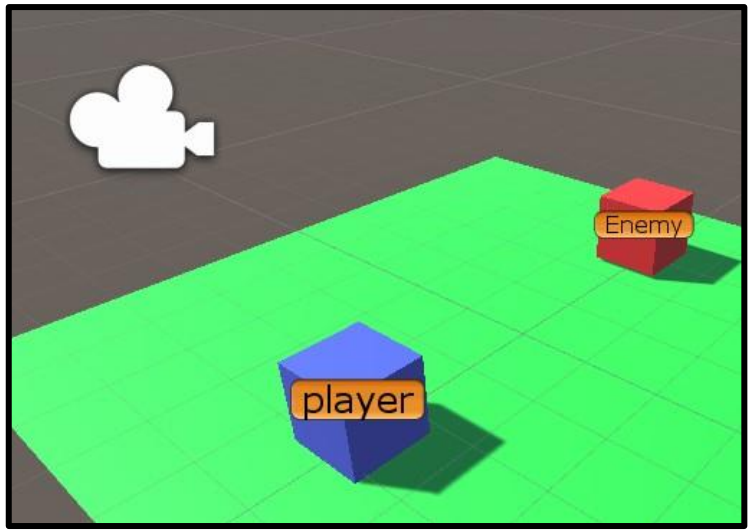
```
static void Main(string[] args) {
    IEnumerator testIE = TestIEnumerator();
    while (testIE.MoveNext()) { // MoveNext() 실행 시 직전에 리턴한 코드 다음 부터 실행
        Console.WriteLine(testIE.Current); // current: 반환된 값
    }
}
static IEnumerator TestIEnumerator() {
    yield return 1;
    yield return 2;
    yield return 3;
}
```

https://waterbeetle.tistory.com/4?category=807747

예제 구현




```
public class Enemy : MonoBehaviour
{
    public enum EnemyState { idle, chase, attack };   // 열거형 선언
    public EnemyState enemyState = EnemyState.idle;   // 변수 선언 및 초기화
    public Transform player;
    void Start()
    {
        StartCoroutine(CheckEnemy());
        StartCoroutine(EnemyAction());
    }
```



```
IEnumerator CheckEnemy()
{
    while (true)
    {
        yield return new WaitForSeconds(0.05f);

        float dist = Vector3.Distance(player.position, transform.position);

        if (dist < 2)
        {
            enemyState = EnemyState.attack;
        }
        else if (dist < 5)
        {
            enemyState = EnemyState.chase;
        }
        else
        {
            enemyState = EnemyState.idle;
        }
    }
}
```

```
IEnumerator EnemyAction()
{
    while (true)
    {
        switch (enemyState)
        {
            case EnemyState.chase:
                print("chase");
                break;
            case EnemyState. attack:
                print("attack");
                break;
            default:
                print("idle");
                break;
        }
        yield return null;
    }
}
```

필수 과제

- 인터페이스, 델리게이트 기초 강의영상(4시간) 주소: https://url.kr/q4nv2l
- 인터페이스, 델리게이트관련 동영상을 듣고 강의 예제와 유사한 자신의 예제 프로그램을 하나씩 작성하시오.
- 제출방법 : Email (abback@mjc.ac.kr)로 다음주 화요일(27일)까지 제출