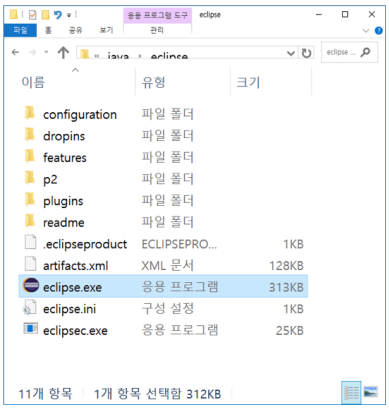
스레드 (Thread)

1. 프로세스 스레드

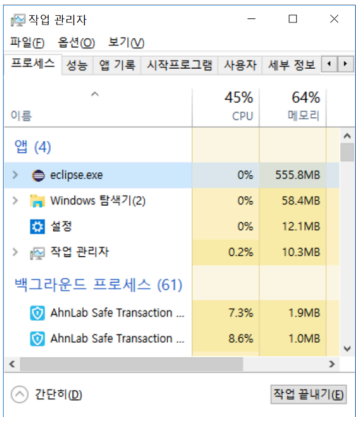
프로세스와 스레드에 대해서 정확히 이해하기 위해서는 메모리 계층구조와 Process에 대해서 알아야한다.

1-1. Program과 Process

▶ 프로그램 : 실행 가능한 파일(HDD, SSD)



▶ 프로세스 : 실행 중인 프로그램(메모리)



- **보조 기억 장치(HDD, SSD)에 저장되어 있는 프로그램(코드의 모음)을 실행하면 주 기억장치(RAM)에 프로세스를 생성하여 CPU가 실행하는 구조**
  - 보조기억장치(HDD, SSD)
    - CPU가 직접 접근하지 못한다. device controller등을 통해 접근 가능하다.
    - 용량이 크며, 속도가 비교적 느리다.
    - 비휘발성 메모리 (컴퓨터를 꺼도 데이터 유지)
    - 정적인 텍스트 (코드)
  - 주기억장치(RAM)
    - CPU가 직접 접근하여 데이터를 사용할 수 있다.
    - 용량이 작으며, 속도가 매우 빠르다.
    - 휘발성 메모리 (컴퓨터를 끄면 데이터 소멸)
    - 동적인 상태 (변수등의 값이 바뀐다.)

> 프로그램 : 보조 기억 장치(HDD, SSD)에 저장되어 있는 프로그램(코드의 모음)
>
> 프로세스 : 프로그램이 실행되어 RAM에 적재되어 실행 중인 상태

1-2. 메모리계층구조

> 컴퓨터에는 왜 메모리가 이렇게 나뉘어져 있는 것일까??
>
> 그냥 보조 기억 장치(HDD, SSD)에 다 때려박으면 될 일 아닌가?
>
> - 지역적 특성때문이다.

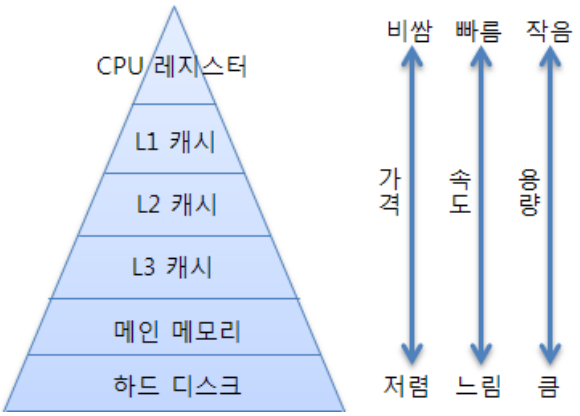
> 우리가 도서관에가서 원하는 여러 권의 책을 제일 빠르게 읽는 방법은?

책장에서 읽고싶은 여러 권의 책을 한번에 내 책상으로 가져와서 읽는게 제일 빠르다.

- 책장 : HDD/SSD (여러권의 책들을 저장할 수 있지만, 찾는데 오래 걸린다.)
- 책상 : RAM (적은 수의 책들을 저장할 수 있지만, 찾는데 엄청 빠르다.)
- 사람 : CPU

> 컴퓨터도 똑같아. 지역적 특성때문에 메모리의 계층을 나누어놓았다.

CPU의 ALU(연산장치)가 요구하는 피 연산자의 데이터가 만약 하드디스크에 있다면 데이터를 가져오기 위해서 L1캐시부터 하드디스크까지 모든 계층을 거쳐야 하고, 다시 하드디스크에서 L1캐시까지 거쳐서 데이터를 가져오게 된다.



- 캐시 메모리가 필요한 이유
  - CPU의 성능이 좋아서 연산 속도가 아무리 빨라진다 해도 연산에 필요한 데이터가 CPU의 레지스터로 옮겨지기까지 기다려야한다. 자주 사용하는 데이터를 캐시에 저장하면 CPU가 데이터를 빨리 처리할 수 있게되기 때문이다.
- Trade - Off
  - 캐시메모리는 속도가 엄청 빠른 대신 용량이 작다.
  - 하드디스크는 속도가 엄청 느린 대신 용량이 크다.

> 결과적으로 속도를 빠르게 하기 위해서 각자의 역할을 하는 메모리가 존재하는 것이다.

> 하드디스크를 HDD에서 SSD로 바꾸면 게임의 로딩 속도가 빨라진다.

> 그 이유는 게임에서의 로딩이 바로 프로그램(HDD/SDD)를 프로세스(SSD)로 바꾸는 과정이기 때문이다.

1-3. Process

> 이해하고 넘어가야 할 부분
>
> 1. 어떻게 여러 개의 프로세스가 동시에 실행되는가?
>   - 현대의 사람들은 컴퓨터를 이용해서 음악을 들으면서 코딩도 하고 웹서핑도한다.
>   - 하나의 CPU로 어떻게 동시에 실행할 수 있는거지?
> 2. Process는 RAM에 어떤 식으로 저장되는가?
>   - CPU가 프로세스들을 왔다갔다 실행한다면 프로세스마다 상태를 어떻게 저장하지?
>   - CPU가 A프로세스를 실행하다가 B프로세스를 실행하고 다시 A프로세스를 실행한다면 A프로세스의 어디부터 실행되는가?

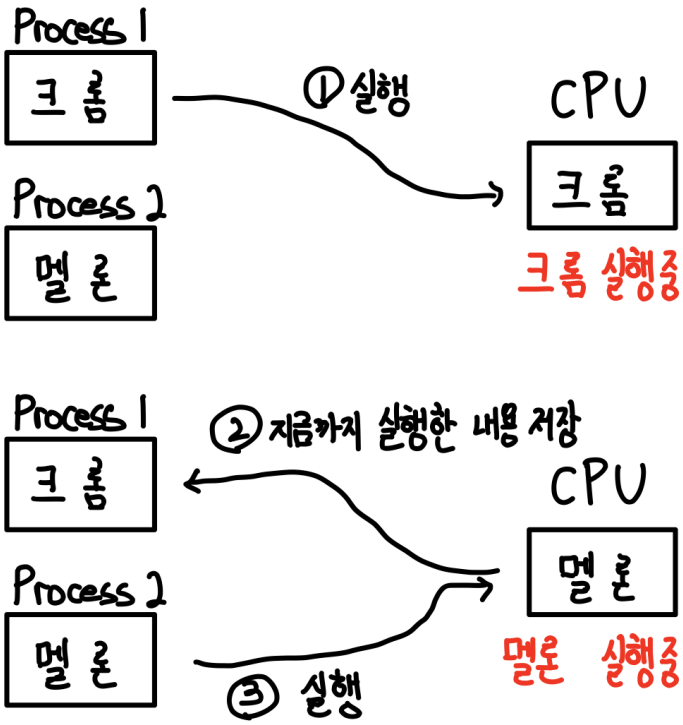
CPU가 프로세스를 처리하는 방법 = 병행

컴퓨터 내의 가장 핵심적인 역할을 하는 중앙 처리 장치인 CPU는 사람의 뇌와 같다.

사람이 두 가지의 생각을 동시에 하지 못하듯, CPU도 동시에 두 개의 프로세스를 실행하고 관리하지 못한다.

그렇지만, **컴퓨터는 여러 개의 프로그램이 함께 동작하는것 처럼 프로세스들을 처리한다.**

제한된 CPU가 여러 개의 프로세스들을 동시에 처리할 수 있는 이유는 병행으로 처리하기 때문이다.



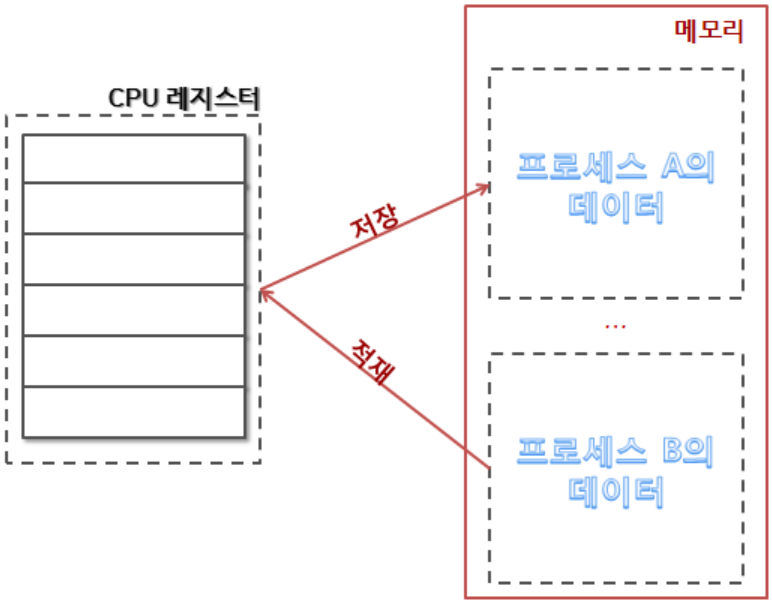
위와 같이 크롬과 멜론은 Time-Sharing(시분할)으로 CPU를 나눠 사용함으로써 사용자가 병행성을 느낄 수 있도록 한다.

위 과정이 너무 빠르게 반복되고 실행되므로 사람들은 이를 알아차리지 못하는 것 뿐이다.

또한, 제한된 CPU의 개수로 여러 프로세스를 처리하기 위해 `CPU 스케줄링`이라는 개념이 나오게 되었다.

> CPU 스케줄링은 내용이 너무 많으므로, 나중에 정리하고자 한다...

컨텍스트 스위칭



여러 프로세스가 서로 경쟁하며 CPU를 점유하여 제어권을 가져가는게 CPU의 처리방법이다.

그리고 **여러 프로세스가 CPU의 제어권을 점유하는 과정에서 점유권을 교환하는 것을 컨텍스트 스위칭이라고 한다.**

> 컨텍스트 (Context)
>
> - 프로세스의 내용(정보)을 의미한다. (PCB = Process Control Block)
>   - 프로세스 식별자 (Process ID)
>   - 프로세스 상태 : 생성, 준비, 실행, 대기, 완료
>   - 프로그램 카운터 (Program Counter) : 이 프로세스가 다음에 실행할 명령어의 주소를 가리킨다.
>   - PC라고 하며, 컨텍스트중에서 제일 중요한 부분을 차지한다.
>   - Process A가 CPU의 점유권을 가지고 있다가, Process B에게 넘겨주고, 시간이 지나 다시 Process A가 CPU의 점유권을 가지면 전까지 실행된 코드의 위치를 알아야한다. PC가 바로 이러한 위치 주소를 저장하고 있는 변수이다.
>   - CPU 레지스터 및 일반 레지스터
>   - CPU 스케줄링 정보
>
> 스위칭 (Switching)
>
> - 교환

1-4. 동기 비동기 블록 논블록

스레드와 관련된 내용이며, 꼭 알고 넘어가야 할 내용이다.

> 참고 사이트 (Good 블로그)
>
> - 동기와 비동기, 그리고 블럭과 넌블럭
> - Blocking-NonBlocking-Synchronous-Asynchronous
> - 블럭,논블럭,동기,비동기 이야기

1-4-1-. Blocking / Non-Blocking

> 블록/논블록은 `호출되는 함수`가 바로 리턴하느냐 마느냐가 관심사다.

행위자가 취한 행위 자체가, 또는 그 행위로 인해 다른 무엇이 **막혀버린, 제한된, 대기**하는 상태

- `호출된 함수`가 **자신이 할 일을 모두 마칠 때까지 제어권**을 **계속 가지고**서 `호출한 함수`에게 바로 돌려주지 않으면 Block
- `호출된 함수`가 **자신이 할 일을 채 마치지 않았더라도 바로 제어권**을 **건네주어(return)** `호출한 함수`가 다른 일을 진행할 수 있도록 해주면 Non-Block

1-4-2. Synchronous / Asynchronous

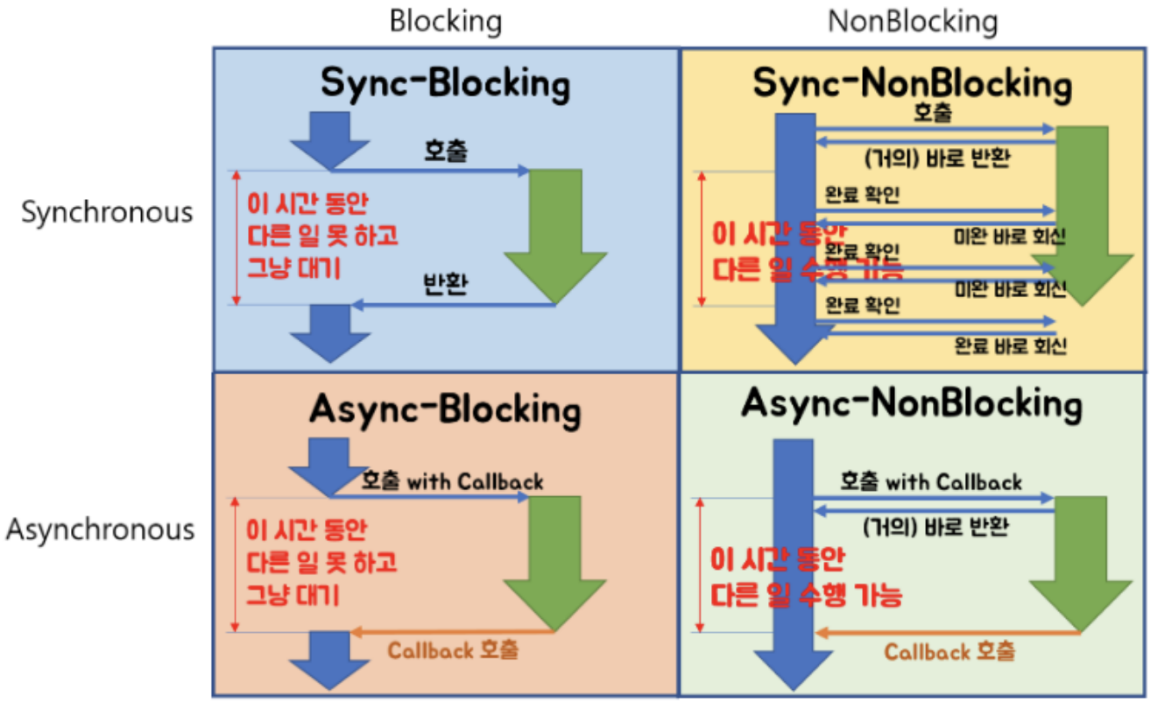
> 동기/비동기는 `호출되는 함수`의 작업 완료 여부를 누가 신경쓰냐가 관심사다.
>
> 행위라고도 본다.

동시에 발생하는 것들 (always plural, can never be singular)

동시라는 것은 시(time)라는 단일계(system)에서 **같이, 함께** 무언가가 이루어지는 두 개 이상의 개체 혹은 이벤트를 의미한다고 볼 수 있다.

- `호출된 함수`의 수행 결과 및 종료를 `호출한 함수`가(`호출된 함수`뿐 아니라 `호출한 함수`도 함께)신경 쓰면 Synchronous
    - A라는 행위와 B라는 행위가 순차적으로 작동한다면 Synchronous
        - A라는 행위가 별개의 것이 아니라, B라는 행위를 관찰하는 행위라면 이것이 동시에 일어나더라도 동기라고 본다.
        - A라는 쓰레드와 B라는 쓰레드가 따로 돌아간다고 해도, 어떤 하나의 행위가 다른 행위에 밀착되어 있다면 두 행위가 다른 쓰레드에서 벌어지더라도 동기를 의미한다.
- `호출된 함수`의 수행 결과 및 종료를 `호출된 함수` 혼자 직접 신경 쓰고 처리한다면(as a callback fn.) Asynchronous
    - A라는 행위와 B라는 행위가 동시(or 순차적이지 않다면)에 실행되고 있으면 Asynchronous
        - A,B 행위 사이에는 인과관계가 있어야한다.
        - 웹서버에서 멀티쓰레드로 각각 A와 B가 다른 클라이언트와 작업 할 때 둘은 동시에 작업하고 있지만, 둘의 인과관계는 없다. 이럴때는 비동기라고 보지 않는다.

1-4-3. 동기/비동기 블록/논블록



Sync-Blocking

A가 실행되다가 B라는 일을 수행하는 함수를 호출해서 B를 시작한다.

B라는 일이 끝나면 함수를 리턴한다.

A와 B는 순차적으로 진행되기 때문에 동기이며, B라는 일을 하는 함수를 호출하고 그 이이 끝나고 나서야 리턴되므로 블럭.

> 서로 같은 일(순차적)을 하며(동기), 제어권을 하나만 가지고 서로 돌려가며 사용한다.(블록)

```
나 : 대표님, 개발자 좀 더 뽑아주세요..
대표님 : 오케이, 잠깐만 거기 계세요!
나 : …?!!
대표님 : (채용 공고 등록.. 지원자 연락.. 면접 진행.. 연봉 협상..)
나 : (과정 지켜봄.. 궁금함.. 어차피 내 일 하러는 못 가고 계속 서 있음)
```

Async-NonBlocking

A는 B의 일을 시작시키고 바로 리턴하므로 논블럭.

그리고 A와 B는 각자 자신의 일을 하므로 비동기.

A와 B는 자신들의 일을 하면서 서로가 제어권을 가지고 있다.

> 서로 다른 일(순차적이지 않음)을 하며(비동기), 제어권을 서로가 따로 가지고 있다.(논블록)

```
나 : 대표님, 개발자 좀 더 뽑아주세요..
대표님 : 알겠습니다. 가서 볼 일 보세요.
나 : 넵!
대표님 : (채용 공고 등록.. 지원자 연락.. 면접 진행.. 연봉 협상..)
나 : (열일중..)
대표님 : 한 분 모시기로 했습니다~!
나 : 😍

// ajax의 댓글이 바로 Async-NonBlocking
```

Sync-NonBlocking

A는 B라는 일을 시킨다. 바로 리턴한다. (논블럭)

B는 일을 하는데, A는 자신의 일을 하지 않는다.

A의 하는 일은 그저 B가 하는 일을 확인하는 것이다.

B가 결과 보고를 했는지를 확인하는 함수를 호출하고, 바로 리턴한다. (논블럭) 즉 결과보고를 받을 때 까지 기다리는 게 아니라, 결과 보고가 나왔는지 확인하고 바로 리턴한다. 이 짓을 계속한다..

즉 함수를 계속 논블럭으로 호출되긴 하나, A는 그저 B를 염탐할 뿐이다. 이 상태를 말한다. 그냥 염탐하지 말고 B가 일을 모두 끝마치고 리턴되길 기다린다.

> 서로 같은 일(순차적)을 하며(동기), 제어권을 서로가 따로 가지고 있다.(논블록)

```
나 : 대표님, 개발자 좀 더 뽑아주세요..
대표님 : 알겠습니다. 가서 볼 일 보세요.
나 : 넵!
대표님 : (채용 공고 등록.. 지원자 연락.. 면접 진행.. 연봉 협상..)
나 : 채용하셨나요?
대표님 : 아직요.
나 : 채용하셨나요?
대표님 : 아직요.
나 : 채용하셨나요?
대표님 : 아직요~!!!!!!
```

Async-Blocking

A는 B라는 일을 시킨다. 바로 리턴한다. (논블럭)

B는 일을 시작하고, A도 자신의 일을 한다. **A는 중간에 B라는 일이 하는 중간 결과를 보고 받아서 처리해야 한다.**

A는 B에게 요청을 해서 중간결과를 기다린다. (블록) 요청의 결과를 받고 나서 그 결과를 이용해서 A는 자신의 일을 처리한다.

동시에 B는 또 자신의 일을 동시에 한다. (비동기)

A는 다시 B에게 중간결과를 요청해서 기다린다. (블록)

요청의 결과를 받고 A는 자신의 일을, B는 자신의 일을 한다. 반복한다.

> 제일 애매하다... 의도적으로 사용하진 않는다고 한다.

```
나 : 대표님, 개발자 좀 더 뽑아주세요..
대표님 : 오케이, 잠깐만 거기 계세요!
나 : …?!!
대표님 : (채용 공고 등록.. 지원자 연락.. 면접 진행.. 연봉 협상..)
나 : (안 궁금함.. 지나가는 말로 여쭈었는데 붙잡혀버림.. 딴 생각.. 못 가고 계속 서 있음)

// 중간에 블록되는 지점이 있지만, 그 이전과 이후에는 각자 자신의 일을 한다.
```

2. 스레드의 구현과 실행

자바에서 Thread를 구현하는 방법은 2가지 있다.

- `Thread` 클래스를 상속받는 방법
- `Runnable` 인터페이스를 구현하는 방법

```
// Thread클래스를 상속
class Thread1 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++){
            System.out.println(getName()); // 부모인 Thread의 getName() 호출
        }
    }
}

// Runnable인터페이스 구현
class Thread2 implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 5; i++){
            // Thread.currentThread(); - 현재 실행중인 Thread 실행
            System.out.println(Thread.currentThread().getName());
        }
    }
}

public class Test {

    public static void main(String[] args){
```

```
        Thread1 th1 = new Thread1();

        Runnable r = new Thread2();
        Thread th2 = new Thread(r);

        th1.start();
        th2.start();
    }
}
// 결과
Thread-0
Thread-0
Thread-0
Thread-0
Thread-0
Thread-1
Thread-1
Thread-1
Thread-1
Thread-1
```

- 실행하는 것이 별로 없어서 `동기/블록` 처럼 보이지만, 비동기적으로 실행되고 있다.

2-1. Thread클래스

```
// JAVA API Thread Class
public class Thread implement Runnable{
  /* 중요 멤버 */
  private Runnable target; // run()을 위한 Runnable
  private ThreadGroup group;
  private volatile String name;

  /* 생성자 */
  // 기본 생성자
  Thread(){
    // ThreadGroup, Runnable, name, stacksize
    this(null, null, "Thread-" + nextThreadNum(), 0);
  }

  // Runnable 매개변수로 받는 생성자
  Thread(Runnable target) {
    this(null, target, "Thread-" + nextThreadNum(), 0);
  }
```

```
// name 매개변수로 받는 생성자
Thread(String name) {
  this(null, null, name, 0);
}

// 진짜 생성자 (모두 이 생성자를 호출한다)
Thread(ThreadGroup g, Runnable target, String name, long stackSize){
  // Thread name 초기화
  if(name == null)
    throw new NullPointerException("name cannot be null");
  this.name = name;

  // ThreadGroup 초기화
  ...

  // Runnable 초기화
  this.target = target;
}


/* start */
public synchronized void start() {

  // ThreadGroup에 add
  group.add(this);

  boolean started = false;
  try {
    start0(); // native 메서드
    started = true;
  } finally {
    try {
      if(!started){
        group.threadStartFaild(this);
      }
    } catch (Throwable ignore){
      // do nothing
    }
  }
}

private native void start0(); // C언어로 된 코드

/* run */
public void run() {
  if(target != null){
    target.run(); // 중요! 개발자가 구현한 Runnable 인터페이스의 run()을 실행하는 것.
  }
}
```

```
  /* 주요 메서드 */
  public static native Thread currentThread(); // Thread 반환

  public final String getName(){
    return name;
  }

}
```

- `run()`의 의미
  - `Thread`클래스 안에 `Runnable`인터페이스 `target`변수가 존재한다.
  - `target`은 개발자가 오버라이딩을 통해 구현한다.
    - Thread클래스 상속 : 상속받은 클래스의 `run()`을 실행한다.
    - Runnable인터페이스 구현 : 구현된 클래스의 `run()`을 실행한다.
  - `run()`은 그저 개발자가 구현한 메서드를 실행할 뿐 새로운 스레드를 만들진 않는다.
- `Thread`를 `start()`하면 해당 스레드는 어떠한 `ThreadGroup`안에 들어간다.
- `currentThread()` : 현재 실행중인 쓰레드의 참조를 반환한다. (native)

> 책의 내용을 외우는 것보다는 Java API와 직접 코드를 분석하여 공부하면 훨씬 쉽게 다가온다.

2-2. Runnable인터페이스

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

- `Runnable`을 사용하면 다중상속이 제한된 자바에서 다른 상속을 받을 수 있게 해준다.

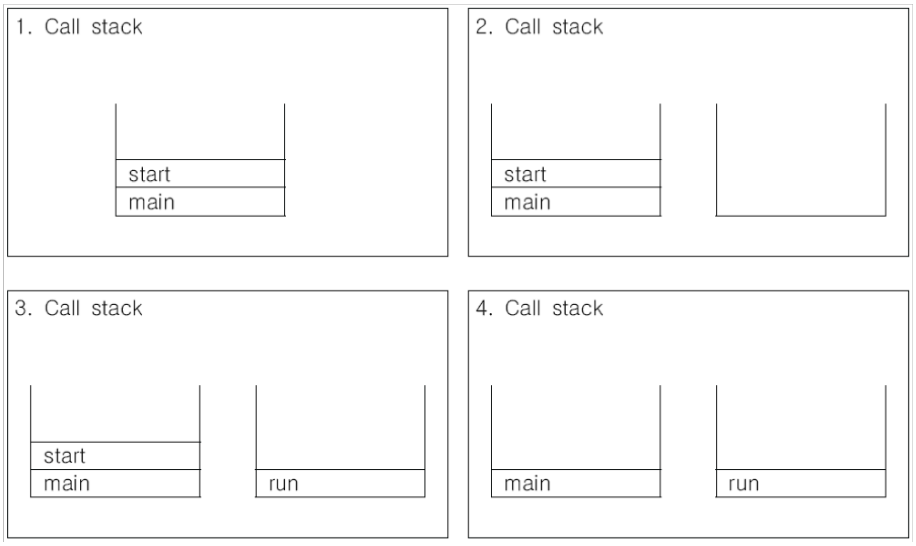
2-3. start()와 run()

```java
public synchronized void start() {
    /**
     * This method is not invoked for the main method thread or "system"
     * group threads created/set up by the VM. Any new functionality added
     * to this method in the future may have to also be added to the VM.
     *
     * A zero status value corresponds to state "NEW".
     */
    if (threadStatus != 0)
        throw new IllegalThreadStateException();

    /* Notify the group that this thread is about to be started
     * so that it can be added to the group's list of threads
     * and the group's unstarted count can be decremented. */
    group.add(this);

    boolean started = false;
    try {
        start0();
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed( t: this);
            }
        } catch (Throwable ignore) {
            /* do nothing. If start0 threw a Throwable then
              it will be passed up the call stack */
        }
    }
}

private native void start0();
```

- `start()`를 실행하면 해당 `Thead`객체를 `ThreadGroup`에 넣는다.

```java
class ThreadTest {
    public static void main(String args[]){
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

```java
class MyThread extends Thread {
    public void run() {
        //...
    }
}
```



- `start()`는 새로운 스레드가 작업을 실행하는데 필요한 호출스택(call stack)을 생성한 다음에 `run()`을 호출한다.
- 생성된 호출스택(call stack)에 `run()`이 첫 번째로 올라가게 된다.

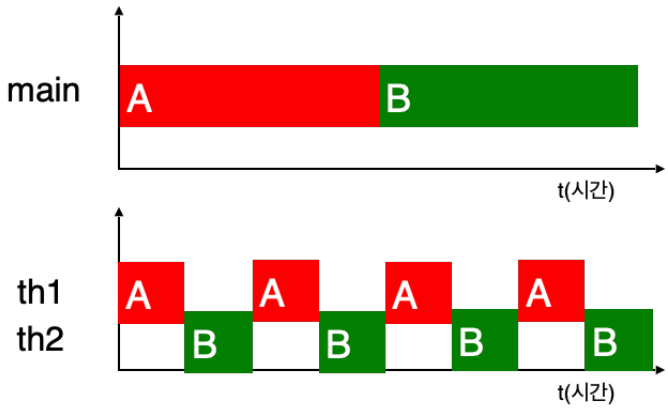
> 모든 스레드는 독립적인 작업을 수행하기 위해 자신만의 Call Stack을 필요로 하기 때문에, 새로운 스레드를 생성하고 실행시킬 때마다 새로운 Call Stack이 생성되고 스레드가 종료된 작업에 사용된 Call Stack은 소멸된다.

> 한 번 실행이 종료된 스레드는 다시 실행할 수 없다.
>
> 다시 실행하고 싶으면 새로운 스레드 객체를 생성해서 실행해야한다.

> 실행 중인 사용자 스레드가 하나도 없을 때 프로그램은 종료된다.

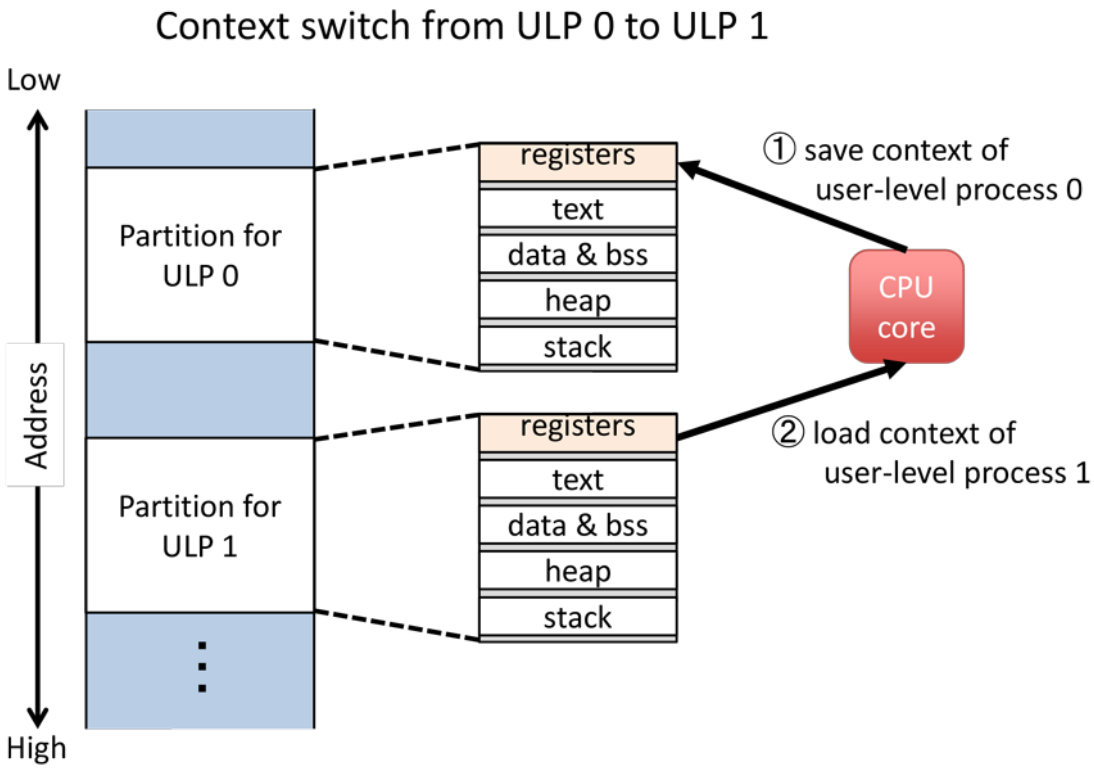
3. 싱글스레드와 멀티스레드

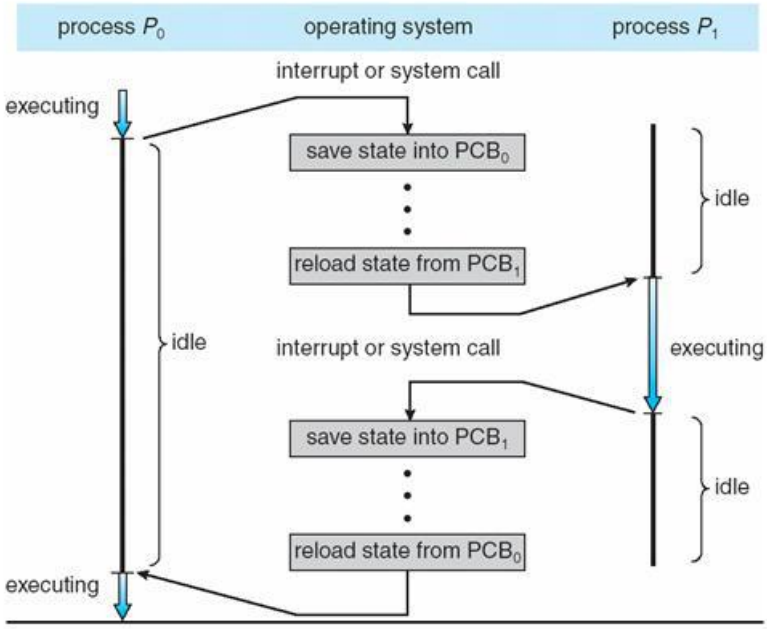
3-1. 싱글스레드와 멀티스레드



- 하나의 스레드로 두 작업을 처리하는 경우는 한 작업 마친 후에 다른 작업을 시작한다. (동기/블록)
- **두 개의 스레드로 작업 하는 경우에는 짧은 시간동안 2개의 스레드(`th1`, `th2`)가 번갈아 가면서 작업을 수행해서 동시에 두 작업이 처리되는 것과 같이 느끼게 된다.**
    - **여러 프로세스들이 CPU의 점유를 가져가려 경쟁하는 것과 같이 여러 스레드가 프로세스의 점유를 가져가려 경쟁한다.**
        - **스레드도 프로세스 안에서 컨텍스트 스위칭을 한다는 의미.**
        - 스레드의 레지스터는 **호출스택의 지역변수**와 **Program Counter(실행위치)**다.

3-2. 프로세스 관점에서의 스레드

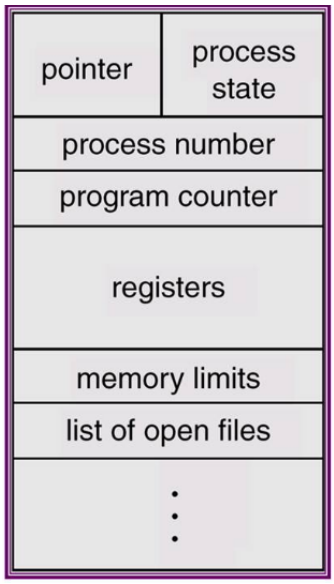




- OS는 개별 프로세스가 다른 프로세스의 영역을 침범하여 오류를 범하지 않도록 **프로세스를 격리시킨다.**
- **CPU가 프로세스A를 실행중에 I/O처리나 시스템 콜 등 인터럽트가 걸리면 실행중이던 프로세스 A의 Context를 저장하고, 실행할 프로세스의 Context를 로드하여 실행한다. (컨텍스트 스위칭)**

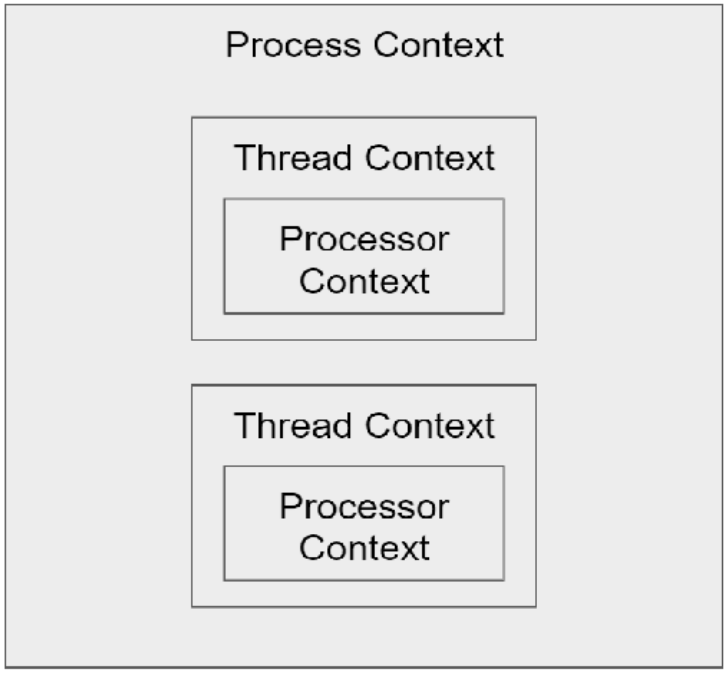
3-2-1. Context

> 컨텍스트 스위칭의 컨텍스트(context)에는 어떠한 내용이 들어있을까?
>
> - PCB (Process Control Block)라고도 한다.



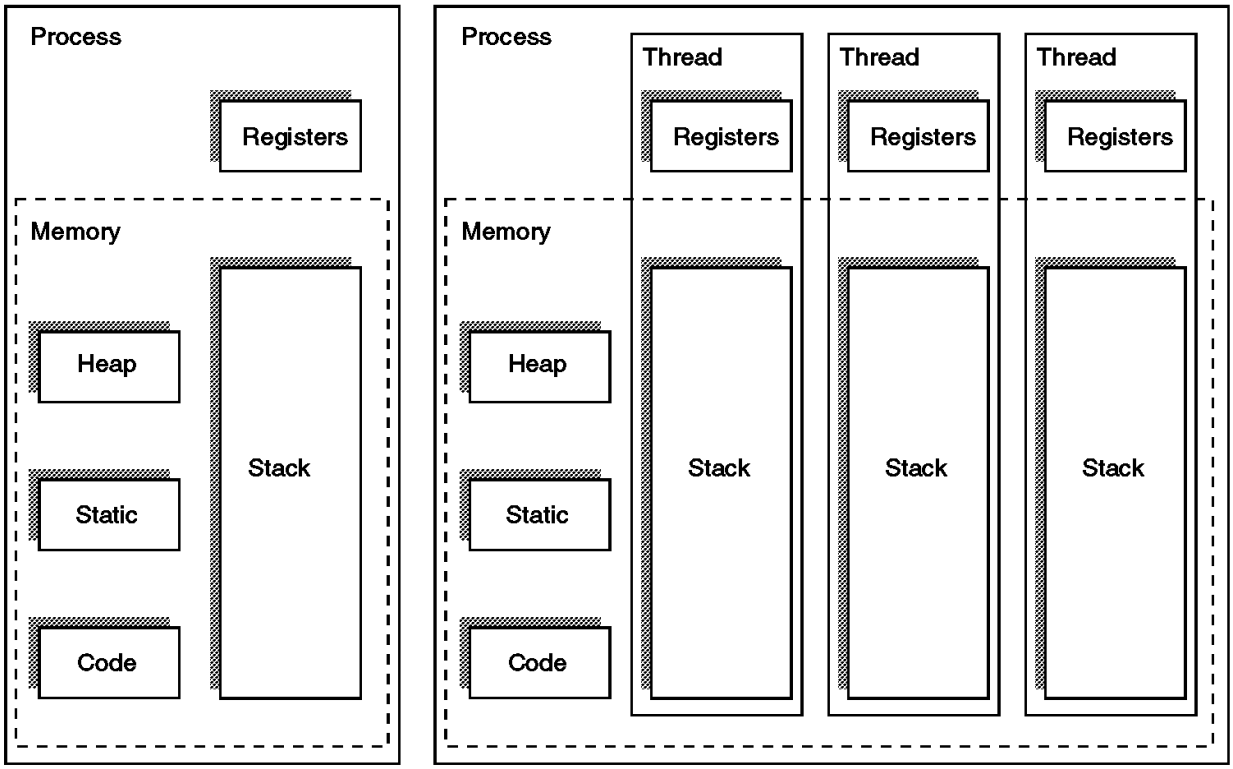
- **process state : 프로세스 상태 (Create, Ready, Running, Waiting, Terminated)**
- process number(PID) : 프로세스 식별자
- **program counter(PC) : 다음 실행할 명령어의 주소값. (지금까지 실행한 다음 위치)**
- registers : 레지스터

3-2-2. 프로세스는 스레드를 포함한다.



- Process Context가 여러 개의 Thread Context를 포함하고 있다.
    - **Process를 실행하면 프로세스내의 스레드 스케줄링에 따라서 스레드가 실행되는 것이다.**

3-3. 메모리 관점에서의 스레드

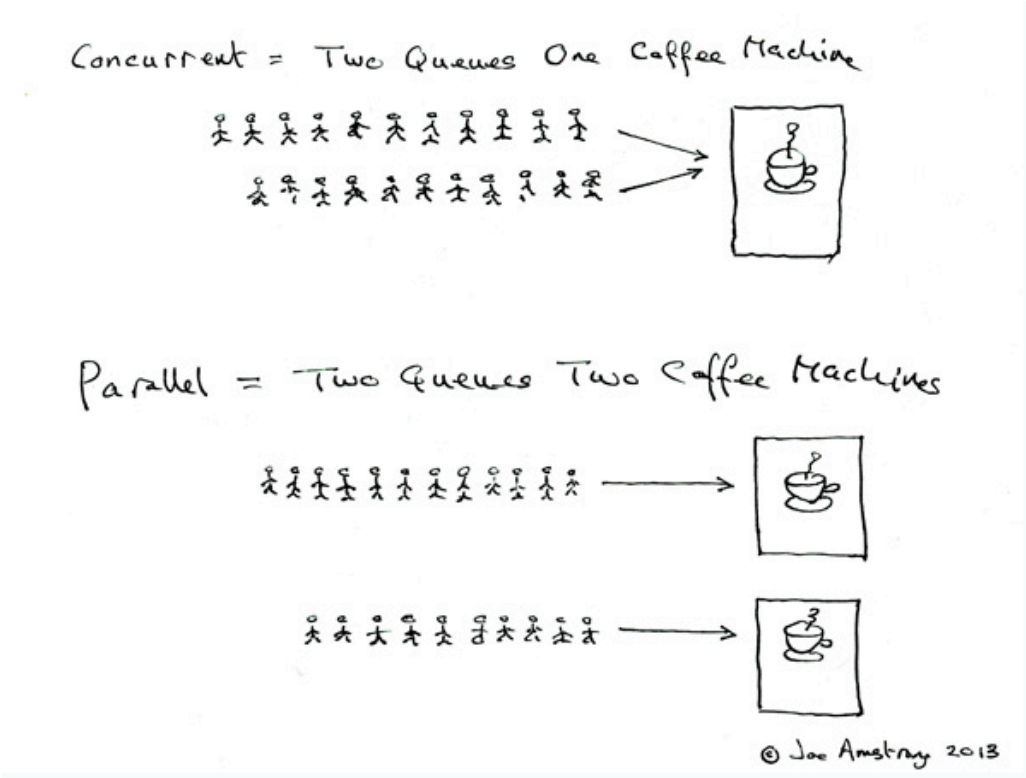


- 프로세스는 프로세스마다의 독립적인 메모리를 가지고 있다.
- **스레드는 프로세스의 Heap, Static, Code 영역을 공유한다.**
    - 스레드 간의 데이터 공유비용이 매우 적다.

> 프로세스 컨텍스트 스위칭시에는 모든 데이터 영역을 저장하고 로드하기 때문에 시간비용이 많이 발생한다.
>
> 스레드 컨텍스트 스위칭은 같은 프로세스 내에서 호출스택의 지역변수와 PC(실행위치)만을 저장하고 로드하기 때문에 시간비용이 훨씬 적다.

3-4. 병행과 병렬

3-4-1. 병행과 병렬의 차이

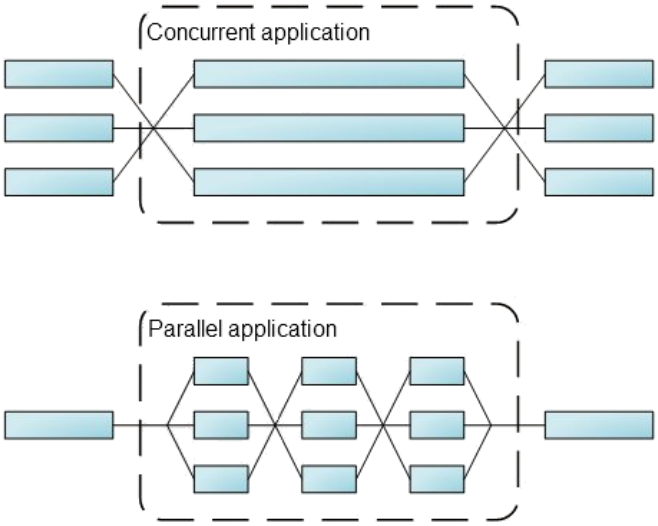


- 병행성(Concurrency) - CPU
  - **동시에 실행되는 것처럼 보이는 것.**
    - 1코어 멀티 스레딩.
  - Single Core
    - 물리적으로 병렬이 아닌 순차적으로 동작할 수 있다.
    - 실제로 Time-Sharing(시분할)으로 CPU를 나눠 사용함으로써 사용자가 병행성을 느낄 수 있도록 한다.
  - Multi Core
    - 물리적으로 병렬로 동작할 수 있다.
    - 각 CPU가 병행처리를하며, 전체적으로 봤을때는 병렬로 동작하는 것.
  - Case
    - Mutex, Deadlock
- 병렬성(Parallelism) - GPU
  - **실제로 동시에 작업이 처리가 되는 것.**
    - n개의 코어, n개의 스레드.
  - 오직 Multi Core에서만 가능하다.
  - Case
    - CUDA

> 병행 : 하나의 작업을 작은 단위로 분할하여 처리한다.
>
> - 싱글코어CPU에서는 순차로 처리된다.
> - 멀티코어CPU에서는 각 CPU끼리 병렬로 처리되며, CPU에 할당된 작업들끼리 순차처리된다.

> 병렬 : 복수의 업무를 동시에 처리 한다.

> 처리관점에서의 병행과 병렬



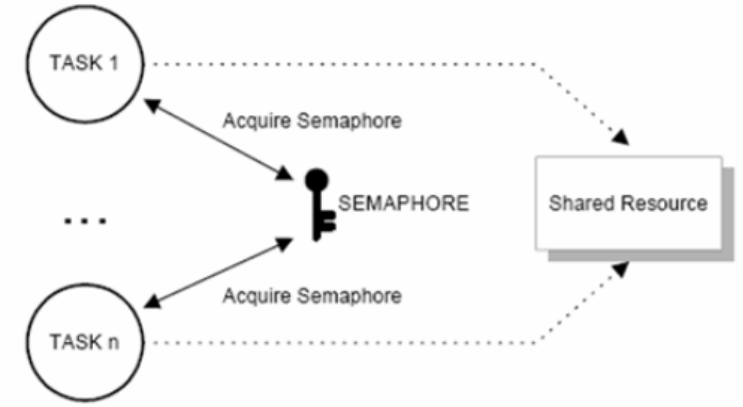
- **Concurrent applications** tend to create a thread that handles a whole series of tasks. Most of the time these concurrent applications create threads because they need an isolated process for a concurrent event.
    - 10개의 업무를 2종류의 독립된 작업으로 분할해 처리하는 것.
- **Parallel applications** divide a process into small tasks that are executed on seperate threads. Because the tasks are small, the threads can be divided evenly over the processors, resulting in very efficient use of a multi-core CPU.
    - 10개의 업무를 2명이 분담하여 처리하는 것.

3-4-2. 스레드의 개수가 2배면 처리량도 2배일까?

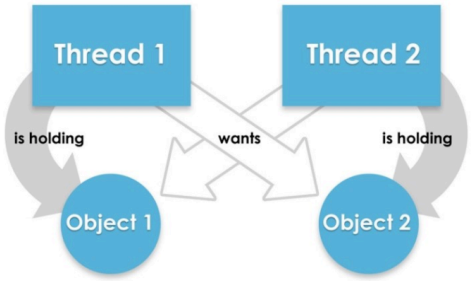
No! 두 가지의 이유가 있다.

1. **Context Switching (문맥 교환)**
    - 스레드 개수가 늘어났다고 해도, CPU 개수에 제약이 있기 때문에 병렬로 동작하지 않는다.
    - 병행적으로 처리하기때문에, 스레드간의 문맥교환의 비용이 발생한다.
2. **Mutual Exclusion (상호 배제)**
    - 만약 스레드가 병렬적으로 처리된다고 해도 스레드 간의 상호배제를 수행하기 위해 오버헤드가 발생한다.
        - 스레드간의 상호 배제의 비용이 발생한다.

> **Mutual Exclusion**

- 공유자원을 어느 시점에서 단지 한 개의 프로세스만이 사용할 수 있도록 하며, 다른 프로세스가 공유자원에 대하여 접근하지 못하게 제어하는 기법.
    - 공유 자원 : 프로세스의 Heap,
- 임계 구역(공유 자원)을 어느 시점에서 한 개의 프로세스만이 사용할 수 있도록 하며, 다른 프로세스가 현재 사용 중인 임계 구역에 대해 접근하려고 할 때 금지하는 행위.
    - 어느 한 스레드가 프로세스의 공유 데이터를 갱신하는 동안 다른 모든 스레드에서 접근하지 않고 우선 대기한다.
    - 그리고 실행중인 스레드가 공유 데이터를 사용하는 작업을 마치고 나서야 시스템이 대기 중인 프로세스 중 하나가 공유 데이터를 접근할 수 있게 해야한다.
    - DeadLock이 발생할 확률이 높다.
- 해결 방법
    - 동기화 - 세마포어, 모니터기법
    - 

DeadLock

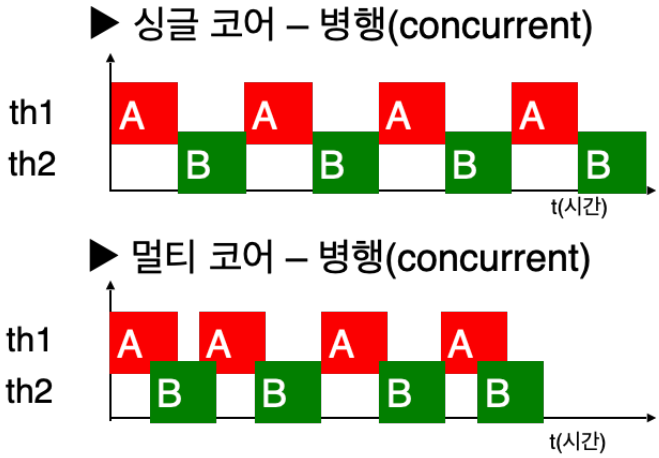
- 교착상태(DeadLock)는 상호 배제에 의해 나타나는 문제점으로, 둘 이상의 프로세스들이 자원을 점유한 상태에서 서로 다른 프로세스가 점유하고 있는 자원을 요구하며 무한정 기다리는 현상을 말한다.
- 결과적으로 서로 상대방의 작업이 끝나기만을 기다리고 있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 만한다.
    - 사다리
        - 아래에 있는 사람은 위로 올라가려고 하고, 위에 있는 사람은 아래로 내려 오려고만 한다면, 두 사람은 서로 상대방이 사다리에서 비켜줄 때 까지 하염없이 기다리고 있을 것이고 결과적으로 아무도 사다리를 내려오거나 올라가지 못하게 된다.
    - 

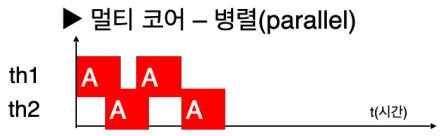
3-4-3. 병행과 병렬 예제

> 싱글스레드


```
public class Test {

    public static void main(String[] args){
        long startTime = System.currentTimeMillis();

        // A
        for(int i = 0; i < 100; i++){
            System.out.printf("%s",new String("-"));
        }

        System.out.println("소요시간1 : " + (System.currentTimeMillis() -
startTime));

        // B
        for(int i = 0; i < 100; i++){
            System.out.printf("%s",new String("|"));
        }

        System.out.println("소요시간2 : "+ (System.currentTimeMillis() -
startTime));
    }
}

// 결과
----------------------------------------------------------------------------------
----------------------소요시간1 : 42
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||||||||||||||||||||||소요시간2 : 70
```

> 멀티스레드



> OS의 JVM마다 코어와 스레드를 다루는 것이 다르기때문에, 이게 싱글코어로 돌린건지 멀티코어로 돌린건지는 모르겠다..
>
> - **싱글코어** : 프로세스의 점유를 받기 위해 스레드 간 컨텍스트 스위칭을 계속해서 한다.
> - **멀티코어** : A스레드는 코어1에서, B스레드는 코어2에서 처리하며, 서로 번갈아가면서 처리한다.
>   - 멀티코어기 때문에 **두 개의 작업**을 분리해서 **병렬적(가짜 병렬)으로 처리한다고 봐도 된다.**
>   - **만약 하나의 작업을 멀티 코어로 처리한다면 진정한 병렬 (찐 병렬)**
>   - 

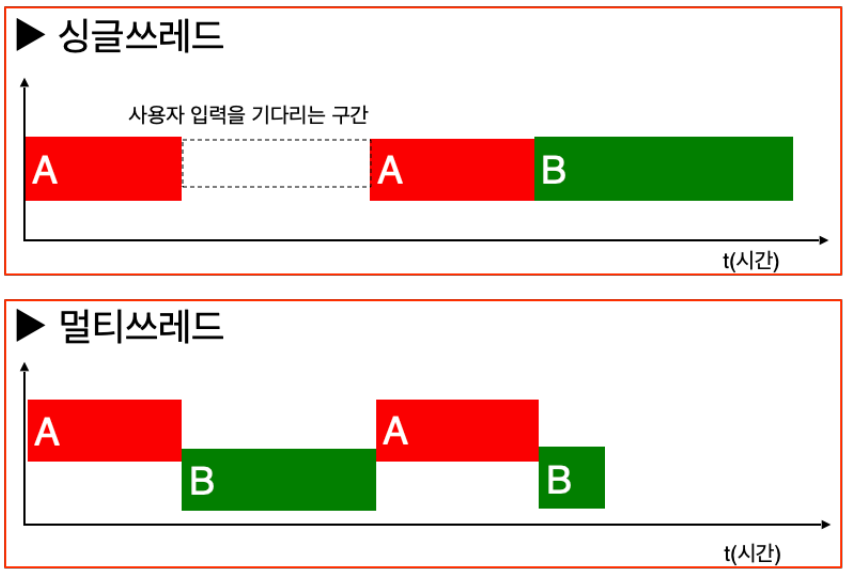

```
class ThreadTest extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.printf("%s", new String("|"));
        }
        System.out.println("소요시간2 : "+ (System.currentTimeMillis() -
Test.startTime));
    }
}


public class Test {

    static long startTime = 0;

    // 스레드 A
    public static void main(String[] args){
        // 다른 스레드 실행 B
        Thread th1 = new ThreadTest();
        th1.start();

        startTime = System.currentTimeMillis();
```

```
        for(int i = 0; i < 100; i++){
            System.out.printf("%s",new String("-"));
        }

        System.out.println("소요시간1 : " + (System.currentTimeMillis() -
startTime));
    }
}
// 결과 ( 싱글코어 )
----------------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||||||||||||||||||||----------------------------------------------------------
----------------------------소요시간2 : 58
소요시간1 : 74

// 결과 ( 멀티코어 )
--------------|||||||||--------|||||||||||-------------------------------
------------|||||||||||||||||||||||||||||||||-------||--------------||||----
-------|||--------------
```

- 두 스레드가 번갈아가면서 작업을 처리하기 때문에 스레드간의 **작업전환시간**이 소요되기 때문에 싱글스레드보다 느릴 수 있다.
- 한 스레드가 화면에 출력하고 있는 동안 다른 스레드는 **출력이 끝나기를 기다려야하는데**, 이때 발생하는 대기시간때문에도 느릴 수 있다. (**모니터 락**)
    - **화면을 잡기 위해서 기다리는 것을 DeadLock라고한다.**

> **싱글코어와 멀티코어의 출력 결과 비교**
>
> - 싱글코어 : 병행
> - 멀티코어 : 병렬 (여러 코어에서 병행적으로 실행되므로 병렬)

3-4-4. Blocking



> Blocking 예제

```
public class Test {

    public static void main(String[] args){
        // 입력을 받는다.
        Scanner sc  = new Scanner(System.in);
        System.out.print("입력 값 : ");
        String input = sc.next();
        System.out.println("입력하신 값은 : "+input);

        // 10부터 1까지 출력한다.
        for(int i = 10; i > 0; i--){
            System.out.println(i);
            try {
                Thread.sleep(1000); // 1초간 지연한다.
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
// 결과
입력 값 : 10
입력하신 값은 : 10
10
9
8
7
6
5
4
3
2
1
```

- 입력값을 받기전까지 for문은 돌아가지 않는다. (블록)

> Non-Blocking 예제

```
class ThreadTest extends Thread {
    @Override
    public void run() {
        for(int i = 10 ; i > 0; i--){
            System.out.println(i);
            try {
                Thread.sleep(1000);
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {

    public static void main(String[] args){
        // 다른 스레드 실행
        Thread th1 = new ThreadTest();
        th1.start();

        // 입력을 받는다.
        Scanner sc  = new Scanner(System.in);
        System.out.print("입력 값 : ");
        String input = sc.next();
        System.out.println("입력하신 값은 : "+input);
    }
}
// 결과
10
입력 값 : 9
8
7
30
입력하신 값은 : 30
6
5
4
3
2
1
```

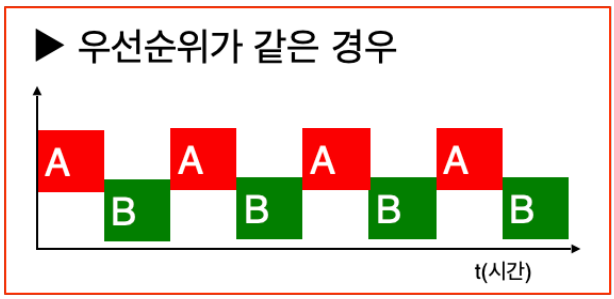
- 입력값을 받기 전부터 for문이 실행되고 있다.

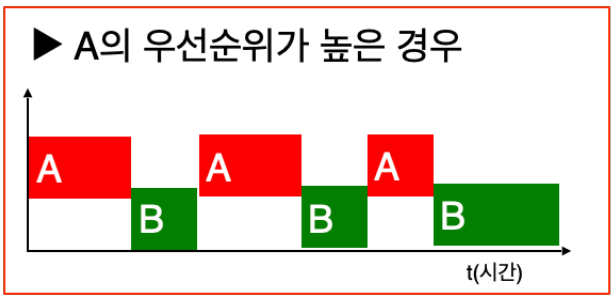
4. 스레드 우선순위

```
public class Thread implements Runnable {
  ...
  /* 우선순위 관련 멤버 */
  public static final int MIN_PRIORITY = 1;
  public static final int NORM_PRIORITY = 5;
  public static final int MAX_PRIORITY = 10;
  ...

  /* 우선순위 관련 메서드 */
  void setPriority(int newPriority);
  int getPriority();
  ...
}
```

- 스레드는 우선순위(Priority)라는 속성(멤버변수)를 가지고 있으며, 이 우선순위의 값에 따라 쓰레드가 얻는 실행시간이 달라진다.

> **스레드는 큐를 사용하는데 어떻게 우선순위에 따라 정렬하는가?? `Priority Queue`와의 관계는?**
>
> - 스레드의 우선순위는 해당 스레드가 CPU(실행)를 점유했을때, 더 오래 잡고 있는다는 의미
> - 스레드의 Priority Queue는 CPU를 점유하고 있지 않은 대기 상태인 스레드 사이에서의 우선순위를 정하는 의미.
>
> **토론이 필요하다!!**





- 우선순위가 같은 경우 각 스레드에게 거의 같은 양의 실행시간이 주어지지만, 우선순위가 다르다면 우선순위가 높은 th1에게 상대적으로 th2보다 더 많은 양의 실행시간이 주어지고 결과적으로 작업 A가 B보다 더 빨리 완료될 수 있다.
  - 단, **멀티코어에서는 병렬처럼 실행되기때문에 우선순위에 따른 차이가 거의 없다.**

> 예제

```
class ThreadTest1 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.print("-");
```

```java
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class ThreadTest2 extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.print("|");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Test {

    public static void main(String[] args){

        Thread th1 = new ThreadTest1();
        Thread th2 = new ThreadTest2();

        th2.setPriority(7);
        System.out.println("Priority of th1 : "+th1.getPriority());
        System.out.println("Priority of th2 : "+th2.getPriority());

        th1.start();
        th2.start();
    }
}
// 결과
Priority of th1 : 5
Priority of th2 : 7
-|-|-|-|-|-||-|-|-|--|-||--|-|-|-||-|--|-|-|-|-|-||-|--|-|-|-|-|-|-|-|-||-|-|-
|--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
```

- `|` 를 출력하는 `th2` 의 우선순위를 7로 높인 예제
    - 싱글코어의 결과

```
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||---
--|||||||||||||||||||||||||||||||||||||-------------------------------
------------------------------------------------
```

  - 싱글코어의 경우 우선순위가 더 높은 `th2`에게 더 많은 실행시간이 주어진다.

- 멀티코어의 결과

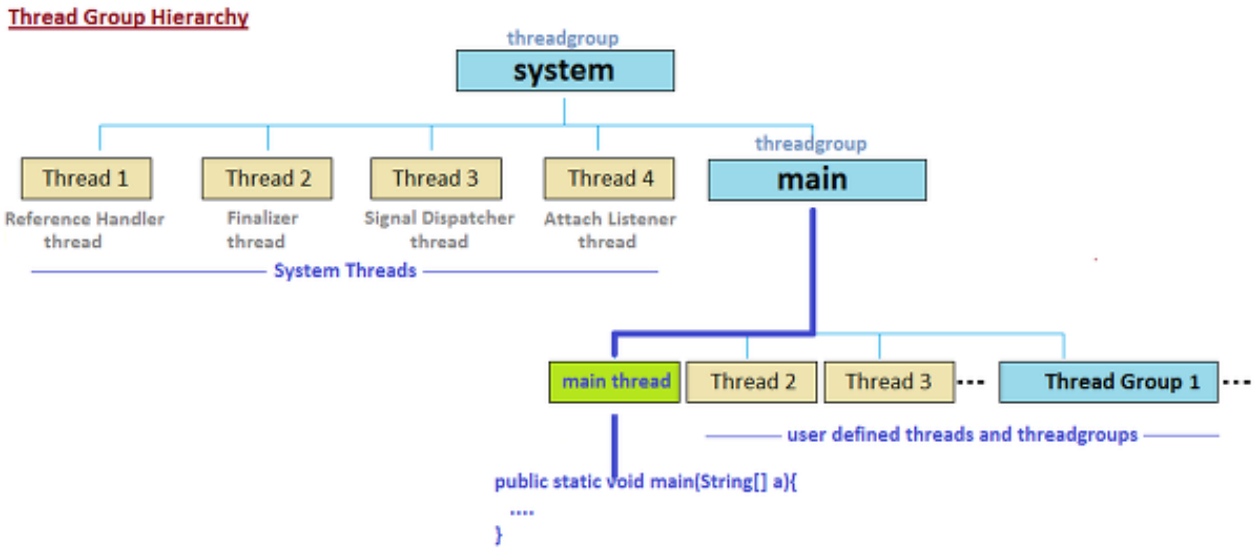
```
-|-|-|-|-|-||-|-|-|--|-||--|-|-|-||-|--|-|-|-|-|-||-|--|-|-|-|-|-|-|-
|-||-|-|-|--|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-
|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|-|
```

  - 멀티코어의 경우 병렬(멀티의 병행)적으로 돌아가기 때문에 우선순위의 따른 차이가 없다.

> **OS의 JVM마다 CPU와 스레드의 스케쥴링이 다르기 때문에 스레드에 우선순위를 부여하는 대신 작업에 우선순위를 두어 `PriorityQueue`에 저장해 놓고, 우선순위가 높은 작업이 먼저 처리되도록 하는 것이 나을 수 있다.**

5. 스레드 그룹

- **서로 관련된 스레드를 그룹으로 묶어서 관리한다.**
  - 폴더시스템과 유사하다. 폴더 안에 폴더 생성 등. 스레드가 파일.
- **보안상의 이유로 도입된 개념이다.**
  - 자신이 속한 스레드 그룹이나 하위 스레드 그룹은 변경할 수 있지만, 다른 스레드 그룹의 스레드를 변경할 수 없다.
- **모든 스레드는 반드시 스레드 그룹에 포함되어야한다.**
  - 스레드 그룹을 지정하지 않고 생성한 스레드는 'main스레드 그룹'에 속한다.
- **자신을 생성한 스레드(부모 스레드)의 그룹과 우선순위를 상속받는다.**



- JVM은 자바 프로그램을 실행하면 `main`과 `system`이라는 스레드 그룹을 만들고 JVM운영에 필요한 스레드들을 생성해서 이 스레드 그룹에 포함시킨다.
  - `main`메서드 (main스레드)는 main스레드 그룹에 속한다.
  - 가비지컬렉션을 수행하는 `Finalizer`스레드는 `system`스레드 그룹에 속한다.
- **우리가 생성하는 모든 스레드는 자동적으로 main스레드 그룹에 속하게 된다.**

6. 데몬 스레드

> 데몬의 사전적의미와 유래
>
> - 도깨비나 유령을 뜻하는 데몬(daemon)이란 이름은 MIT의 MAC 프로젝트 프로그래머들이 만들었다.
>   - 보이지 않는 곳에서 분자들을 골라주는 일을 하고 있는 유령에서 영감을 얻었다고 한다.
>   - 그리스 신화에서도 신들이 관여하지 않는 일들을 처리하는 데몬이 등장한다.
> - 데몬이란 이름이 나오고 유닉스 시스템이 이 용어를 받아들여 사용함으로써 데몬은 사용자가 직접 신경쓰지 않도록 하면서 백그라운드에서 일을 처리해 주는 것을 의미하게 되었다.

6-1. 스레드의 종류

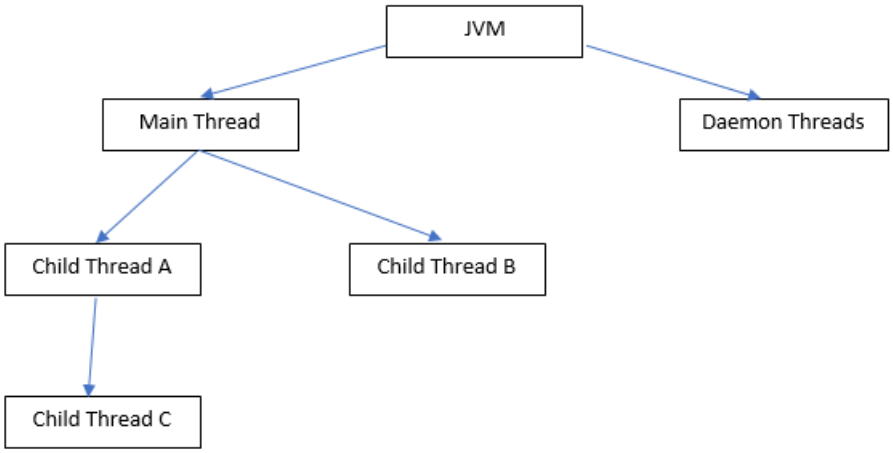
스레드는 크게 일반 스레드와 데몬 스레드로 나눌 수 있다.

- 일반 스레드 : 데몬 스레드가 아닌 스레드.
- 데몬 스레드 : 일반 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드
    - 일반 스레드가 모두 종료되면 데몬 스레드는 강제적으로 자동 종료된다.
    - 예제
        - 가비지 컬렉션, 워드프로세서의 자동저장, 화면자동갱신

> **일반 스레드가 생성한 스레드는 일반 스레드, 데몬 스레드가 생성한 스레드는 데몬스레드이다.**

6-2. 자바 메인 스레드

> Main thread is the most important part of any application. Whenever we run an application the main thread is executed. A main thread is needed so that we can spawn or create the child threads. So, we can create child threads through main thread and start them. The main thread is the last thread to finish the execution i.e., the main thread terminates the program. Typically, JVM starts the main thread and other daemon threads at the same time. So, when an application is started the main thread and other daemon threads are started by JVM. The main thread can further start multiple child threads and the child threads can start further threads. 출처 : Multithreading in Java



- **JVM이 시작되면 생성되는 모든 스레드는 메인 스레드를 제외한 모두가 데몬 스레드이다.**
  - 가비지 컬렉션등등
- 메인 스레드가 종료되면 가비지 컬력센등 데몬 스레드들은 종료된다.
- 개발자가 작성한 모든 코드는 메인 스레드에 존재하기때문에 설정해주지 않는 한 일반 스레드이다.

6-3. 데몬스레드 예제

```
public class Test implements Runnable{

    static boolean autoSave = false;

    public static void main(String[] args){
        Thread th = new Thread(new Test());
        th.setDaemon(true); // 데몬스레드 설정. ( 이 부분이 없으면 종료되지 않는다. )
        th.start();

        for(int i = 1; i <= 10; i++){
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) { e.printStackTrace(); }
            System.out.println(i);

            if(i == 5)
                autoSave = true;
        }

        System.out.println("프로그램을 종료합니다.");
    }

    @Override
    public void run() {
        while(true) {
            try {
```

```
                Thread.sleep(3*1000); // 3초마다
            } catch (InterruptedException e) { e.printStackTrace(); }

            if(autoSave){
                autoSave();
            }
        }
    }

    public void autoSave() {
        System.out.println("작업파일이 자동저장되었습니다.");
    }
}
```

- 메인 스레드
  - 1초마다 숫자를 출력
  - 5초가 되면 `autoSave = true`
- 데몬 스레드
  - 3초마다 `autoSave`의 값을 확인후, `true`면 `autoSave()`호출

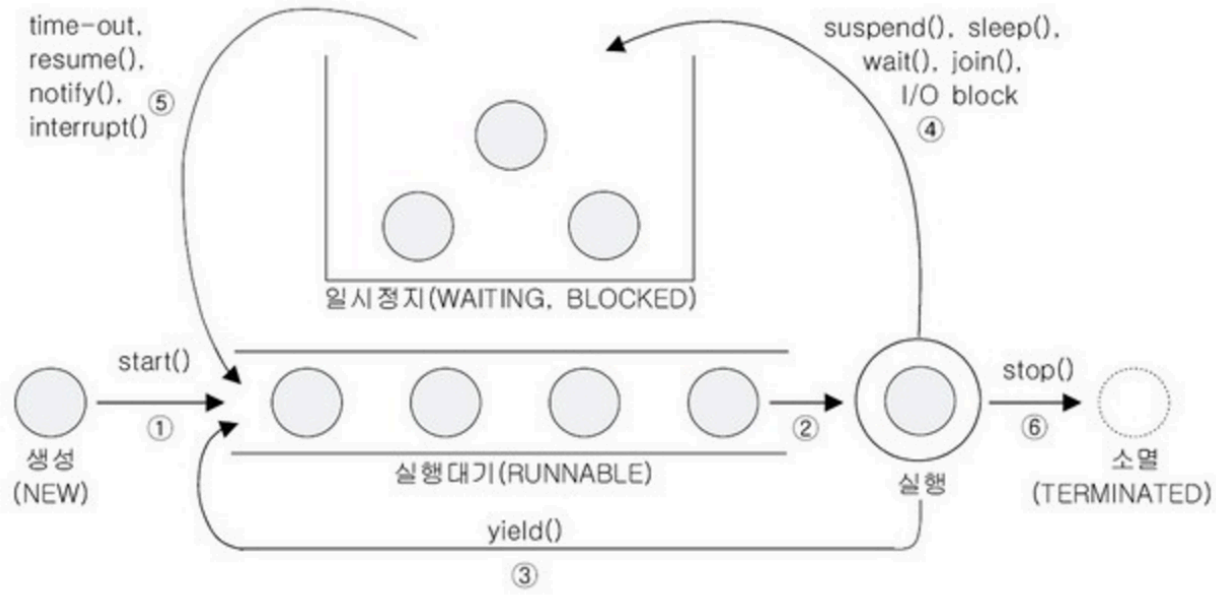
> **`main`스레드가 `th`라는 스레드를 실행시켰다. 그리고 `th`스레드를 데몬 스레드로 설정했다.**
>
> - `main`스레드는 `th`의 부모스레드.
> - 데몬 스레드는 부모 스레드가 죽으면 자기도 죽는다. 효도를 잘한다:)

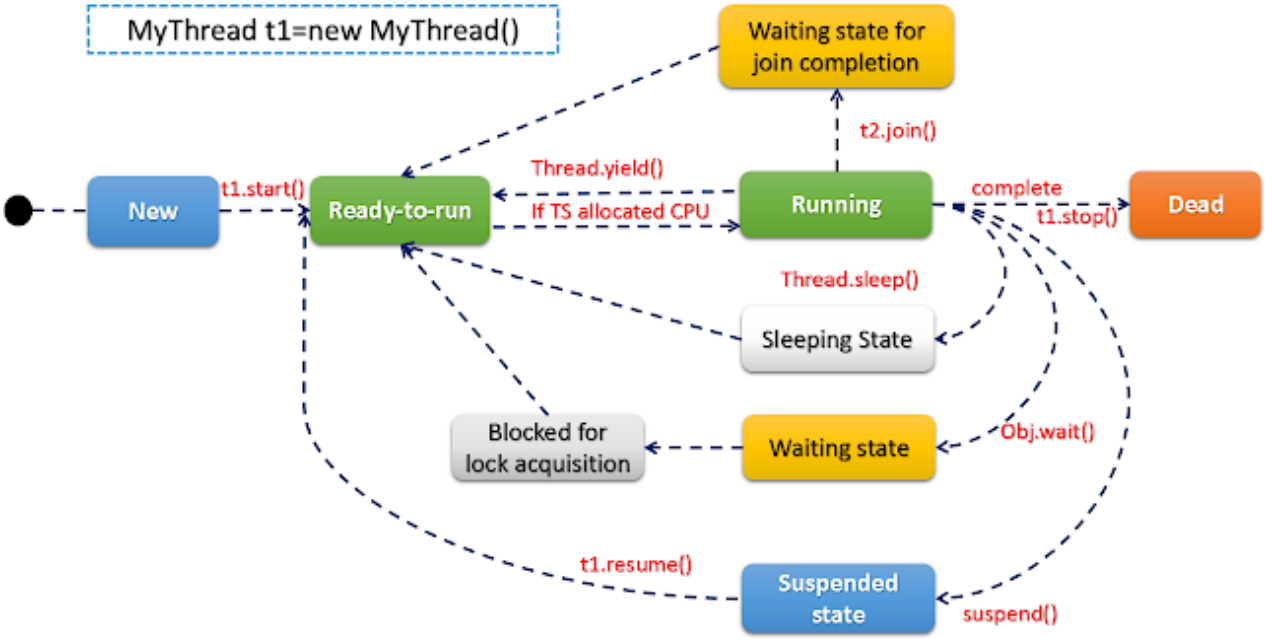
7. 스레드의 실행제어

효율적으로 자원과 시간을 멀티 스레드 환경에서 사용하기 위해서는 "동기화"와 "스케줄링"이 필수적이다.

7-1. 스레드 생명주기

스레드는 효율적인 "동기화"와 "스케줄링"을 위해 생명주기를 갖는다.





- 쓰레드를 생성하고 `start()`를 호출하면 바로 실행되는 것이 아니라 실행대기열(Runnable)에 저장되어 자신의 차례가 될 때까지 기다려야 한다. 실행대기열은 큐(queue)와 같은 구조로 먼저 실행대기열에 들어온 스레드가 먼저 실행된다.
- 실행대기상태에 있다가 자신의 차례가 되면 실행상태(Running)가 된다.
- 주어진 실행시간이 다되거나 `yield()`를 만나면 다시 실행대기상태가 되고 다음 차례의 스레드가 실행상태가 된다.
- 실행 중에 `suspend()`, `sleep()`, `wait()`, `join()`, `I/O block`에 의해 일시정지상태가 될 수 있다.
  - `I/O block`은 입출력작업에서 발생하는 지연사태를 말한다.
  - 사용자의 입력을 기다리는 경우에 일시정지 상태에 있다가 사용자가 입력을 마치면 다시 실행대기상태가 된다.
- 지정된 일시정지기간이 다 되거나(`time-out`), `notify()`, `resume()`, `interrupt()`가 호출되면 일시정지 상태를 벗어나 다시 실행대기열에 저장되어 자신의 차례를 기다리게 된다.

- 실행을 모두 마치거나 `stop()`이 호출되면 스레드가 소멸된다.

7-2. 스레드 실행제어

메서드	설 명
static void sleep(long millis) static void sleep(long millis, int nanos)	지정된 시간(천분의 일초 단위)동안 쓰레드를 일시정지시킨다. 지정한 시간이 지나고 나면, 자동적으로 다시 실행대기상태가 된다.
void join( ) void join(long millis) void join(long millis, int nanos)	지정된 시간동안 쓰레드가 실행되도록 한다. 지정된 시간이 지나거나 작업이 종료되면 join( )을 호출한 쓰레드로 다시 돌아와 실행을 계속한다.
void interrupt( )	sleep( )이나 join( )에 의해 일시정지상태인 쓰레드를 깨워서 실행대기상태로 만든다. 해당 쓰레드에서는 InterruptedException이 발생함으로써 일시정지상태를 벗어나게 된다.
void stop( )	쓰레드를 즉시 종료시킨다.
void suspend( )	쓰레드를 일시정지시킨다. resume( )을 호출하면 다시 실행대기상태가 된다.
void resume( )	suspend( )에 의해 일시정지상태에 있는 쓰레드를 실행대기상태로 만든다.
static void yield( )	실행 중에 자신에게 주어진 실행시간을 다른 쓰레드에게 양보(yield)하고 자신은 실행대기상태가 된다.

▲ 표 13-2 쓰레드의 스케줄링과 관련된 메서드

> `resume()`, `stop()`, `suspend()`는 스레드를 교착상태(Dead-Lock)로 만들기 쉽기 때문에 deprecated되었다.

7-2-1. sleep(long millis)

```
static void sleep(long millis)             // 천분의 일초 단위
static void sleep(long millis, int nanos) // 천분의 일초 + 나노초
```

- `sleep(long millis)`는 일정시간동안 스레드를 멈추게 한다.

```
try {
    Thread.sleep(1, 500000);     // 쓰레드를 0.0015초 동안 멈추게 한다.
} catch(InterruptedException e) {}
```

- 예외처리를 해야 한다.
  - `sleep()`에 의해 일시정지 상태가 된 스레드는 지정된 시간이 다 되거나 `interrupt()`가 호출되면, `InterruptedException`이 발생되어 잠에서 깨어나 실행대기 상태가 된다.

```
try {
    th1.sleep(2000);
} catch(InterruptedException e) {}
```

```
try {
    Thread.sleep(2000);
} catch(InterruptedException e) {}
```

- 특정 스레드를 지정해서 멈추게 하는 것은 불가능하다.
    - `Thread.sleep()`을 호출한 스레드가 잠에 들게 된다.

7-2-2. interrupt()와 interrupted()

> interrupt의 사전적 의미
>
> - 방해하다, 중단시키다, 차단하다

```
class Thread { // 알기 쉽게 변경한 코드
        ...
    boolean interrupted = false;
        ...
    boolean isInterrupted() {
        return interrupted;
    }

    boolean interrupt() {
        interrupted = true;
    }
}
```

```
void    interrupt()          쓰레드의 interrupted상태를 false에서 true로 변경.
boolean isInterrupted()      쓰레드의 interrupted상태를 반환.
static boolean interrupted() 현재 쓰레드의 interrupted상태를 알려주고, false로 초기화
```

`interrupt()`는 진행중인 스레드의 작업이 끝나기 전에 취소시켜야 할 때가 있는 경우 사용한다.

- 예를 들어 큰 파일을 다운로드 받을 때 시간이 너무 오래 걸리면 중간에 다운로드를 포기하고 취소할 수 있어야 한다.
- `interrupt()`는 스레드에게 작업을 멈추라고 요청한다.
    - 반대로 일시 정지된 스레드를 다시 실행상태로도 요청한다.
- **멈추는 것은 아니고, 그저 스레드의 interrupted 상태 (인스턴스 변수)를 바꾼다.**
    - CPU 프로세스의 인터럽트와 같은 역할
    - I/O 요청이 들어오면 인터럽트가 걸리거나, 시스템 콜에 의해서 인터럽트가 걸리거나

> 예제

```
class ThreadTest extends Thread {
    @Override
    public void run() {
        int i = 10;

        while(i != 0 && !isInterrupted()){
            System.out.println(i--);
            for(long x = 0; x < 2500000000L; x++); // 시간 지연
        }
```

```
        System.out.println("카운터가 종료되었습니다.");
    }
}

public class Test {

    public static void main(String[] args){
        ThreadTest th1 = new ThreadTest();
        th1.start();

        Scanner sc = new Scanner(System.in);
        System.out.print("아무 값이나 입력하세요 : ");
        String input = sc.next();
        System.out.println("입력하신 값은 : "+input+" 입니다.");
        th1.interrupt();
        System.out.println("isInterrupted() : "+th1.isInterrupted());
    }
}
// 결과
10
아무 값이나 입력하세요 : 9
8
7
6
4
입력하신 값은 : 4 입니다.
isInterrupted() : true
카운터가 종료되었습니다.
```

- 사용자가 입력을 하자마자 바로 `th1.interrupt()`가 호출되고 `th1`스레드가 종료되며 모든 스레드가 종료된다.

> 스레드가 `sleep()`, `wait()`, `join()`에 의해 일시정지 상태(WAITING)에 있을 때, 해당 스레드에 대해 `interrupt()`를 호출하면 `sleep()`, `wait()`, `join()`에서 `InterruptedException`이 발생하고 스레드는 `실행대기 상태(Runnable)`로 바뀐다.

```
class ThreadTest extends Thread {
    @Override
    public void run() {
        int i = 10;

        while(i != 0 && !isInterrupted()){
            System.out.println(i--);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // interrupt(); // 꼭 해주어야 멈춘다.
```

```
            }
        }
        System.out.println("카운터가 종료되었습니다.");
    }
}

public class Test {

    public static void main(String[] args){
        ThreadTest th1 = new ThreadTest();
        th1.start();

        Scanner sc = new Scanner(System.in);
        System.out.print("아무 값이나 입력하세요 : ");
        String input = sc.next();
        System.out.println("입력하신 값은 : "+input+" 입니다.");
        th1.interrupt();
        System.out.println("isInterrupted() : "+th1.isInterrupted());
    }
}
// 결과
10
아무 값이나 입력하세요 : 5
입력하신 값은 : 5 입니다.
9
isInterrupted() : true // main의 sout이 먼저 출력되고, th1스레드에서 Exception이 발생한다.
8
7
6
5
4
3
2
1
카운터가 종료되었습니다.
```

- 이전과 다르게 종료되지 않은 이유는 `Thread.sleep(1000)`에서 `InterruptedException`이 발생했기 때문이다.
    - **`sleep()`에 의해 스레드가 잠시 멈춰있을 때, `interrupt()`를 호출하면 `InterruptedException`이 발생**되고 스레드의 **`interrupted`상태는 `false`로 자동 초기화된다.**

7-2-3. suspend(), resume(), stop()

void suspend()	쓰레드를 일시정지 시킨다.
void resume()	suspend()에 의해 일시정지된 쓰레드를 실행대기상태로 만든다.
void stop()	쓰레드를 즉시 종료시킨다.

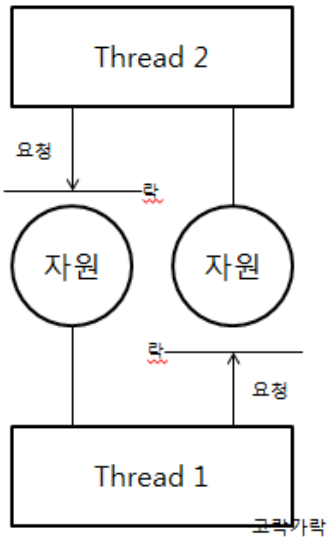
- 스레드의 실행을 일시정지, 재개, 완전정지 시킨다. 교착상태에 빠지기 쉽다.

```
class ThreadEx17_1 implements Runnable {
    boolean suspended = false;
    boolean stopped   = false;

    public void run() {
        while(!stopped) {
            if(!suspended) {
                /* 쓰레드가 수행할 코드를 작성 */
            }
        }
    }
    public void suspend() { suspended = true;  }
    public void resume()  { suspended = false; }
    public void stop()    { stopped = true;    }
}
```

- **`suspend()`, `resume()`, `stop()`은 deprecated되었으므로, 직접 구현해야 한다.**

> **deprecated된 이유는?**
>
> - 제일 큰 이유는 스레드가 사용중이던 자원들이 불안전한 상태로 남겨져 데드락이 걸리기 쉽기 때문이다.
> - `stop()`
>
> ```
> public class Test implements Runnable {
>   static int j = 0;
>   static int i = 100;
>
>   public synchronized void run() {
>     while(i > 0){
>       j++; // stop
>       i--;
>     }
>   }
>
>   public static void main(String[] args) throws
> InterruptedException {
>     Thread thread = new Thread(new Test());
>
>     thread.start();
>     Thread.sleep(5000);
>     thread.stop();
>   }
> }
> ```

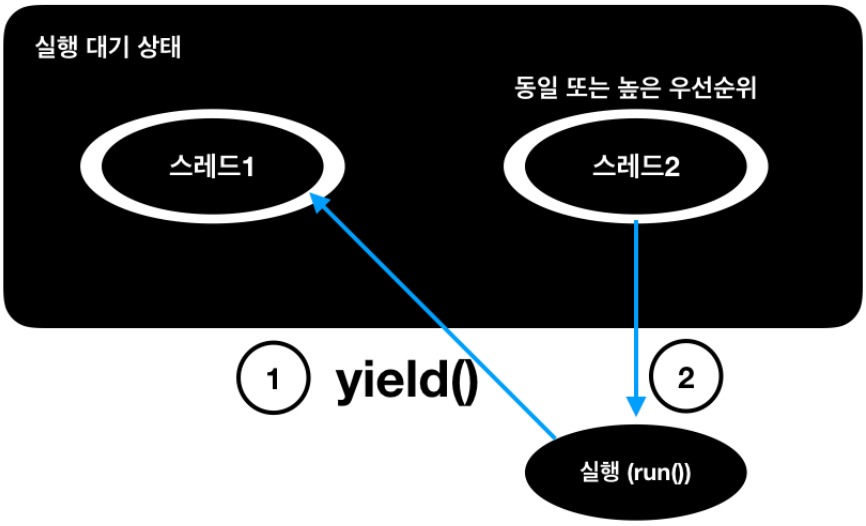
        - while문 안에 두가지 작업이 일어난다. j와 i는 항상 짝이 맞아서 합이 100이여야한다고 가정하자.
        - 하지만 중간에 `Thread.stop()`메서드가 호출되어 `j++`만 실행하고 끝났다고 한다.
        - 이렇게 끝나면 사용자는 어떻게 끝났는지 알수가 없다. 나중에 어떤 에러가 나타날지도 모른다.
        - **`while`문 안의 구문은 통으로 다뤄져야 하며 임계구역(Critical Section)이다.**
- `suspend()`와 `resume()`
    - `suspend()`는 잠시 대기하는 목적인데, `synchronized`를 걸어놓은 메서드(`run()`)가 실행되다 일시정지되면, 다른 스레드에서 공유 자원에 대한 접근이 불가능하다. (위 예제에서는 `i`와 `j`에 접근이 불가능하다.)
    - 만약에 스레드끼리 서로 공유된 자원에 대해 접근하기 위해 `resume()`되어 있지 않은 놈과 서로 엉키다 보면 데드락이 발생하기 쉽다.



    -
        - `Thread1`과 `Thread2`과 서로 같은 자원에 락을 걸며 병행적으로 실행하는 도중, `Thread1`이 `suspend()`에 의해 일시정지되면 락이 걸린 상태로 일시정지상태가 된다.
        - `Thread2`가 CPU를 점유하고 `Thread1`과 같은 자원에 접근하려고보니 락이 걸려 있어 `Thread1.resume()`을 실행하면, 또 다시 `Thread1`이 실행되는데, 이때 `Thread2`도 락이 걸려있어 데드락이 걸리게 된다.

7-2-3. yield()

> yield의 사전적 의미
>
> - 양도하다



- `yield()`는 스레드 자신에게 주어진 실행시간을 다음 차례의 스레드에게 양보(yield)하도록 한다.
  - 스케줄러에 의해 1초의 실행시간을 할당받은 스레드가 0.5초의 시간동안 작업한 상태에서 `yield()`가 호출되면, 나머지 0.5초는 포기하고 다시 실행대기상태가 된다.

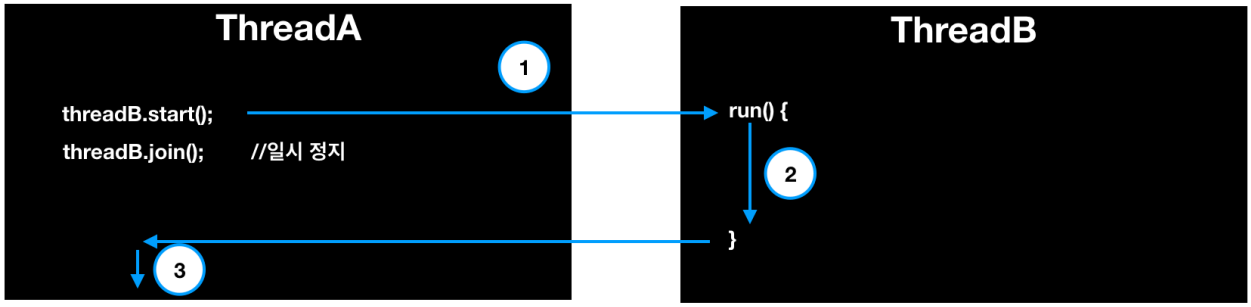
```
class ThreadTest implements Runnable {
    boolean suspended = false;
    boolean stopped = false;

    Thread th;

    public ThreadTest(String name){
        th = new Thread(this, name);  // Runnable인 현재 객체와 이름을 Thread의 생성자로 넘긴다.
    }

    @Override
    public void run() {
        String name = th.getName();

        while(!stopped){
            if(!suspended){
                System.out.println(name);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    System.out.println(name + " - interrupted");
                }
            } else {
                Thread.yield();
            }
        }
        System.out.println(name + " - stopped");
    }
```

```
    public void suspend() {
        this.suspended = true; // 다음번에 하지 말아라
        th.interrupt(); // Sleep상태의 스레드를 바로 멈추기위해서 ( 바로 하지 말아라 )
        System.out.println(th.getName() + " - interrupted() by suspend()");
    }

    public void resume() {
        this.suspended = false;
    }

    public void stop() {
        this.stopped = true;
        th.interrupt();
        System.out.println(th.getName() + " - interrupted() by stop()");
    }

    public void start() {
        th.start();
    }
}

public class Test {

    public static void main(String[] args){
        ThreadTest th1 = new ThreadTest("*");
        ThreadTest th2 = new ThreadTest("**");
        ThreadTest th3 = new ThreadTest("***");
        th1.start();
        th2.start();
        th3.start();

        try {
            Thread.sleep(2000);
            th1.suspend();
            Thread.sleep(2000);
            th2.suspend();
            Thread.sleep(3000);
            th1.resume();
            Thread.sleep(3000);
            th1.stop();
            th2.stop();
            Thread.sleep(2000);
            th3.stop();
        } catch (InterruptedException e){}
    }
}
```

- 분석이 필요하다.

7-2-4. join()

> join의 사전적의미
>
> - 참여하다
> - 자신의 작업 중간에 다른 스레드의 작업을 참여(join)시킨다는 의미.




```
void join()                          // 작업이 모두 끝날 때까지
void join(long millis)               // 천분의 일초 동안
void join(long millis, int nanos)    // 천분의 일초 + 나노초 동안
```

- `join()`는 지정된 시간동안 특정 스레드가 작업하는 것을 기다린다.
  - 스레드 자신이 하던 작업을 잠시 멈추고 다른 스레드가 지정된 시간동안 작업을 수행하도록 할 때 `join()`을 사용한다.
  - 시간을 지정하지 않으면 해당 스레드가 작업을 모두 마칠 때까지 기다리게 된다.
  - 작업 중에 다른 스레드의 작업이 먼저 수행되어야할 필요가 있을 때 `join()`을 사용한다.

```
class ThreadTest extends Thread {

    public ThreadTest(String name){
        this.setName(name);
    }

    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.print(new String(this.getName()));
        }
    }
}

public class Test {

    static long startTime;

    public static void main(String[] args){
        ThreadTest th1 = new ThreadTest("-");
        ThreadTest th2 = new ThreadTest("|");
```

```
        th1.start();
        th2.start();
        startTime = System.currentTimeMillis();

        try {
            th1.join(); // main스레드가 th1의 작업이 끝날 때까지 기다린다.
            th2.join(); // main스레드가 th2의 작업이 끝날 때까지 기다린다.
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("소요시간 : "+(System.currentTimeMillis() -
Test.startTime));
    }
}
// 결과
||||||||||||||||||||||||||||||---------
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||--------
--------------------------------------------------------------------------------
-----소요시간 : 6
```

- `join()`을 사용하지 않았다면 main스레드는 바로 종료되었겠지만, `join()`으로 스레드 `th1`과 `th2`의 작업을 마칠 때까지 main스레드가 기다리도록 했다.

7-2-5. sleep(), yield(), join()의 예제

1. sleep()의 예시
   - 혼자서 프린터를 사용하여 문서 1000장을 복사하는데 너무 지치니까 100장 마다 5분 씩 쉬고 다시 복사를 합니다.
2. yield()의 예시
   - 프린터를 사용하여 문서 1000장을 복사하는데 복사하는 도중, 다른 사람이 사용하려 하면 사용하도록 하고 자신은 잠시 쉽니다.
3. join()의 예시
   - 하나의 내용에 대한 문서가 2000장 입니다. 이 때, 1000장 씩 나눠서 역할을 분담 받았습니다.
     단, 2000장의 문서가 모두 모여야 내용이 완전해지고, 그렇지 않을 경우 의미 없고, 알 수 없는 내용의 문서가 되어 버립니다.
   - 결국 다른 사람이 담당하는 문서의 내용을 합하여 하나의 내용으로 만들어야 하기 때문에
     다른 사람이 담당하는 역할이 끝날 때까지 기다립니다.

> 출처 : https://widevery.tistory.com/28

7-3. 스레드 안전한 종료 방법

> **스레드는 `run()`메서드만 끝나면 자동적으로 종료되므로 `stop()`메서드 사용 대신 `run()`메서드가 정상적으로 종료되도록 코드를 작성해야 한다.**
>
> - 스레드를 안전하게 종료시키는 방법은 스레드의 `run()`메서드가 정상적으로 종료 되도록 유도하는 것이다.

7-3-1. flag

스레드에서 안전하게 끝마쳐야하는 작업을 `while`문으로 감싸고 flag를 설정해놓는다.

```
class ThreadTest extends Thread {

    private boolean flag;

    public void setFlag(boolean flag){
        this.flag = flag;
    }

    @Override
    public void run() {
        while(!flag){
            System.out.println("실행중 !");
        }
        System.out.println("자원 정리");
        System.out.println("실행 종료");
    }
}

public class Test {

    public static void main(String[] args){
        ThreadTest th = new ThreadTest();
        th.start();

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) { e.printStackTrace(); }
        th.setFlag(true);
    }
}
```

- 1초 후에 `th`의 스레드가 안전하게 멈추게 된다.

7-3-2. interrupt() 사용

```
class ThreadTest extends Thread {
    @Override
    public void run() {
        try {
            while(true){
```

```
                System.out.println("실행중");
                Thread.sleep(1); // 1밀리세컨드마다 일시정지 (InterruptedException을
실행시키기위함.)
            }
        } catch (InterruptedException e) {}
        System.out.println("자원 정리");
        System.out.println("실행 종료");
    }
}

public class Test {

    public static void main(String[] args){
        ThreadTest th = new ThreadTest();
        th.start();

        try {
            Thread.sleep(1000);
            th.interrupt();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

- main메서드에서 `th`의 `interrupt()`메서드를 실행하게 되면 `th`가 `sleep()`메서드로 일시 정지 상태가 될 때 `th`에서 `InterruptedException`이 발생하여 예외처리(catch)블록으로 이동한다.
- 결국, `th`는 `while`문을 빠져나와서 `run()`메서드를 정상 종료하게 된다.

> Thread의 `interrupt()` 메소드는 스레드가 일시 정지 상태에 있을 때 `InterruptedException` 예외를 발생시키는 역할을 한다. 이것을 이용하면 Thread의 `run()` 메소드를 정상 종료시킬 수 있다.

> 물론 `sleep()`을 쓰지 않고도 가능하다.

```
class ThreadTest extends Thread {
    @Override
    public void run() {
      // while(!isInterrupted())도 가능 (앞에 this가 생략되었기 때문)
this.isInterrupted().
        while(!(Thread.interrupted())){  // currentThread.interrupted();
            System.out.println("실행중");
        }
        System.out.println("자원 정리");
        System.out.println("실행 종");
    }
}

public class Test {
```

```
    public static void main(String[] args){
        ThreadTest th = new ThreadTest();
        th.start();

        try {
            Thread.sleep(1000);
            th.interrupt();
        } catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

- 스레드 객체의 `interrupt` 멤버를 사용하는 것이다.

> `Interrupted()`와 `isInterrupted()`의 차이
>
> ```
> public static boolean interrupted() { return currentThread().isInterrupted( ClearInterrupted: true); }
>
> /**
>  * Tests whether this thread has been interrupted.  The interrupted
>  * status of the thread is unaffected by this method.
>  *
>  * <p>A thread interruption ignored because a thread was not alive
>  * at the time of the interrupt will be reflected by this method
>  * returning false.
>  *
>  * @return  {@code true} if this thread has been interrupted;
>  *          {@code false} otherwise.
>  * @see     #interrupted()
>  * @revised 6.0
>  */
> @Contract(pure = true)
> public boolean isInterrupted() {
>     return isInterrupted( ClearInterrupted: false);
> }
>
> /**
>  * Tests if some Thread has been interrupted.  The interrupted state
>  * is reset or not based on the value of ClearInterrupted that is
>  * passed.
>  */
> @HotSpotIntrinsicCandidate
> private native boolean isInterrupted(boolean ClearInterrupted);
> ```
>
> - `Interrupted()`는 static메서드 **(native)**
>   - `Thread.interrupted()` : 이 메서드가 호출된 현재 실행중인 스레드의 `interrupt`값이 무엇인지 체크.
>   - 이 메서드를 호출한 스레드의 `interrupt`값 확인
>   - `Interrupted == Thread.currentThread().isInterrupted()`
> - `isInterrupted()`는 인스턴스 메서드
>   - `objThread.isInterrupted()` : `objThread`스레드 객체의 인스턴스 멤버인 `interrupt`값이 무엇인지 체크.
>   - 주어진 어떤 스레드 객체에 대한 인스턴스 멤버 `interrupt`값 확인