멀티스레드와 동시성

2. 프로세스와 스레드 / 3. 스레드 생성과 실행 / 4.5. 스레드 제어와 생명 주기

이한솔 2024.09 1주차

1. 프로세스와 스레드

멀티태스킹과 멀티프로세싱

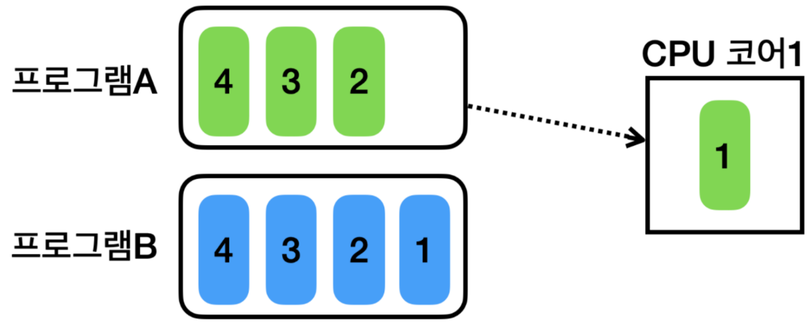


그림1 - 프로그램A의 코드1 실행 시작

- 단일 프로그램 2개(A, B)를 동시 실행을 가정 - CPU의 코어는 1개
  - 프로그램 A 먼저 실행하고 끝난 뒤, 프로그램 B 실행
- 프로그램의 실행: 프로그램을 구성한느 코드를 순서대로 CPU 연산을 수행하는 일
- CPU의 코어는 1개로 가정했으므로 한번에 하나의 프로그램만 처리 가능
- 하지만, 이러한 방식은 사용자에게 불편함을 줄 것

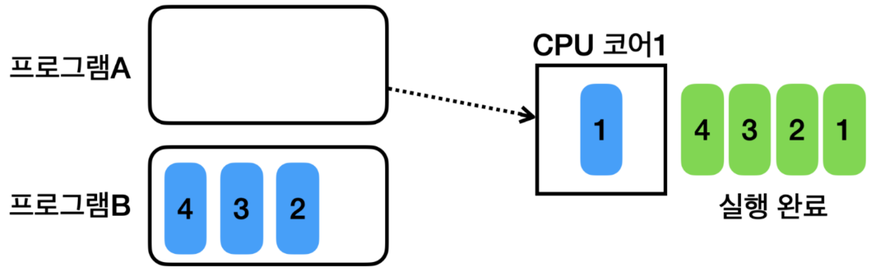


그림4 - 프로그램B 실행 시작

“멀티태스킹”의 등장

하나의 CPU 코어로 여러 프로그램을 동시에 실행

멀티태스킹(Multitasking)

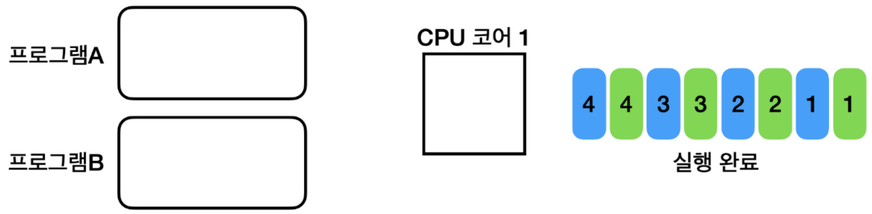


그림3 - 프로그램 수행 완료

- 프로그램 A와 B를 일정한 시간마다 번갈아가며 CPU 연산 처리
  - 시분할 기법: 각 프로그램의 실행 시간을 분할하여 마치 동시에 실행되는 것처럼 하는 방법
- 운영체제에서 배운 프로세스 스케줄링: FCFS, 우선순위, 라운드 로빈 + 선점/비선점 등 알고리즘을 적용하여 CPU를 최대한 사용하도록 최적화

멀티프로세싱(Multiprocessing)

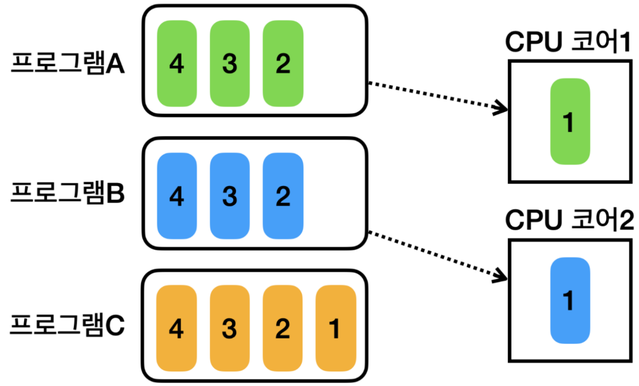


그림1 - 프로그램A, 프로그램B 실행

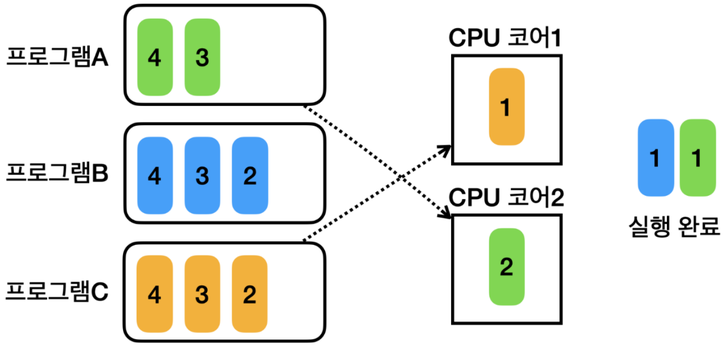


그림2 - 프로그램C, 프로그램A 실행

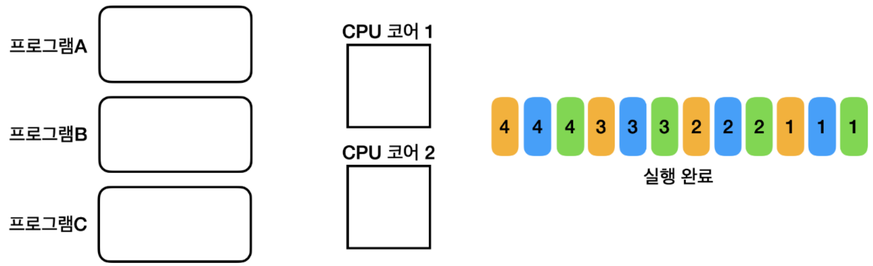


그림3 - 실행 완료

- 프로그램 A, B, C 3개이고 CPU 코어가 2개일 때
  1. 프로그램 A와 B를 실행 - 동시 실행 가능
  2. 프로그램 C와 A 실행 - 처리하던 A, B를 멈추고, 프로그램 C, A를 처리
  3. 각 프로그램들을 번갈아가며 처리하고 완료

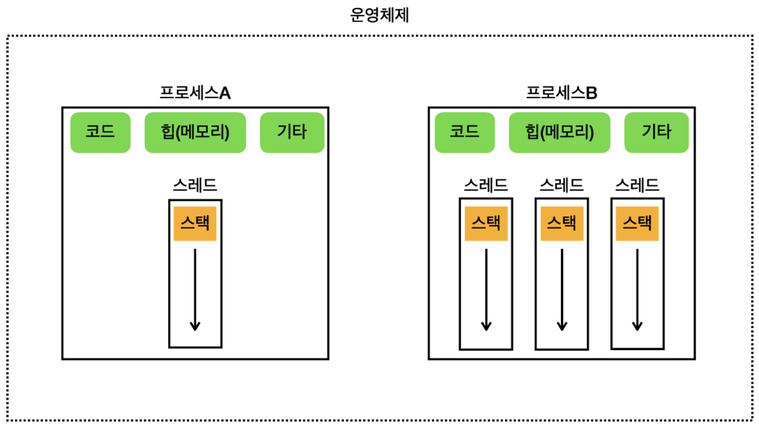
- 멀티프로세싱: 컴퓨터 시스템에서 둘 이상의 프로세서(CPU 코어)를 사용해 여러 작업을 동시에 처리하는 기술, 동시에 더 많은 작업을 가능케 함

- 멀티프로세싱과 멀티태스킹의 비교

  - 멀티프로세싱
    여러 CPU 코어를 사용해 동시에 작업 처리
    하드웨어 기반의 성능 향상, 현대의 컴퓨터 시스템

  - 멀티태스킹
    단일 CPU 코어가 여러 작업을 동시에 수행하는 것처럼 보이게 하는 것
    소프트웨어 기반으로 CPU 연산 시간을 분할해 각 작업에 할당
    현대 운영 체제에서 여러 프로그램들이 동시에 실행되는 환경

프로세스와 스레드



- 프로세스(실행중인 프로그램)
  파일로 존재하던 프로그램을 실행시키고 나서 프로그램의 인스턴스가 생성됨
  프로세스는 각자 독립적인 메모리 공간을 갖고 있음 -> 다른 프로세스에서 간섭 불가능
  ex) 익스플로러의 탭은 스레드로 되어있어 다른 탭에까지 영향을 미침

- 프로세스의 메모리 구조
  - code: 실행할 프로그램의 코드가 저장되는 부분
  - data: 전역 변수, 정적 변수가 저장되는 부분 (그림의 기타 부분)
  - heap: 동적으로 할당되는 메모리 영역
  - stack: 메서드 호출시 생성되는 지역 변수와 반환 주소가 저장되는 영역(스레드)

- 스레드(프로세스는 하나 이상의 스레드(메인스레드)를 반드시 포함함)
  프로세스 내에서 실행되는 작업의 단위
  하나의 프로세스 내에서 여러 개의 스레드가 존재할 수 있음 -> 프로세스 내의 특정 메모리 영역을 공유함 -> 가벼움

- 스레드의 메모리 구성
  공유 메모리: code, data, heap 영역을 공유
  개별 스택: 각 스레드는 자신만의 스택을 가짐

프로세스와 스레드

- 프로그램이 실행된다는 의미
  1. 운영체제는 디스크에 있는 파일인 프로그램을 메모리에 적재하여 프로세스 생성
  2. 프로세스 내의 존재하는 코드를 한줄씩 실행

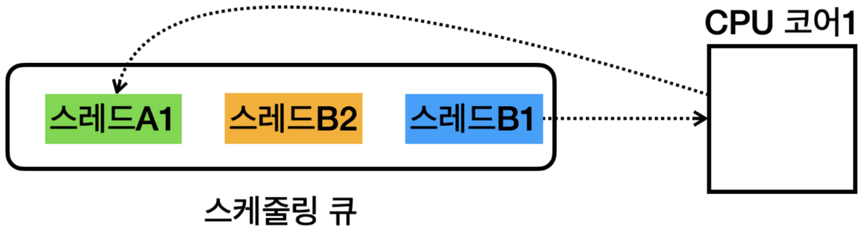
- 코드를 하나씩 실행하며 내려가는 존재 - 스레드 -> **프로세스의 코드를 실행하는 흐름**

- **프로세스 내에 적어도 한개 이상의 스레드를 가짐**

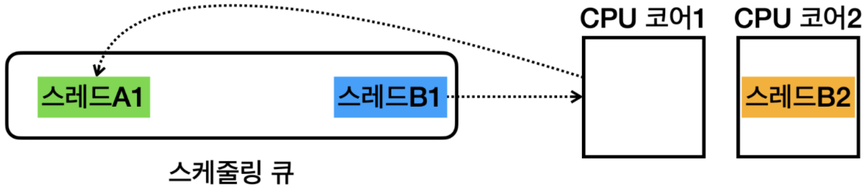
- 즉, 프로세스는 실행 환경과 자원을 제공하는 컨테이너이며 스레드는 CPU를 사용해 코드 하나하나를 실행하는 역할을 함

- 멀티스레드의 필요성 - 하나의 프로그램에도 동시에 여러 작업들이 요구됨
  - ex) 워드 프로그램으로 문서 편집 및 문서가 자동으로 저장되고 맞춤법 검사도 함께 수행됨
    - 스레드1: 문서 편집 / 스레드2: 문서 자동 저장 / 스레드3: 맞춤법 검사

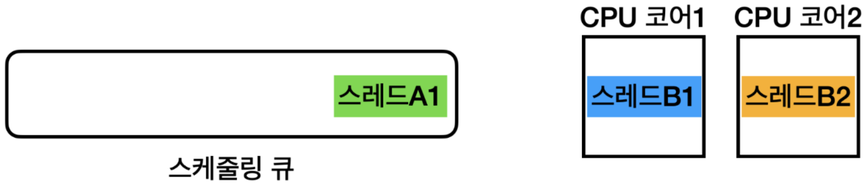
스레드와 스케줄링



- 단일 코어 스케줄링
  내부에 스케줄링 큐를 갖고 있으며 각가의 스레드는 스케줄링 큐에서 대기
  하나씩 큐에서 꺼내 연산 처리하고 다시 스케줄링 큐에 넣음
- 멀티 코어 스케줄링 - 코어 수 만큼 병렬 처리 가능
  스케줄링 큐에 대기중인 스레드들을 가져다 병렬 처리



- 스레드A1의 수행을 잠시 멈추고, 스레드A1을 스케줄링 큐에 다시 넣는다.



프로세스, 스레드와 스케줄링

- 멀티태스킹과 스케줄링
  - 멀티태스킹: 동시에 여러 작업 수행
  - 스케줄링: 멀티태스킹을 위해 운영체제는 스케줄링 알고리즘을 적용하여 적절하게 CPU 시간을 여러 태스크에 나눠 배분
- 프로세스와 스레드
  - 프로세스: 실행중인 프로그램
  - 스레드: 프로세스 내에 실행되는 가장 작은 작업 단위
- 프로세스의 역할
  - 실행 환경 제공
  - 프로세스 자체는 운영체제의 스케줄러에 의해 실행되지 않고 프로세스 내의 스레드가 실행됨

컨텍스트 스위칭

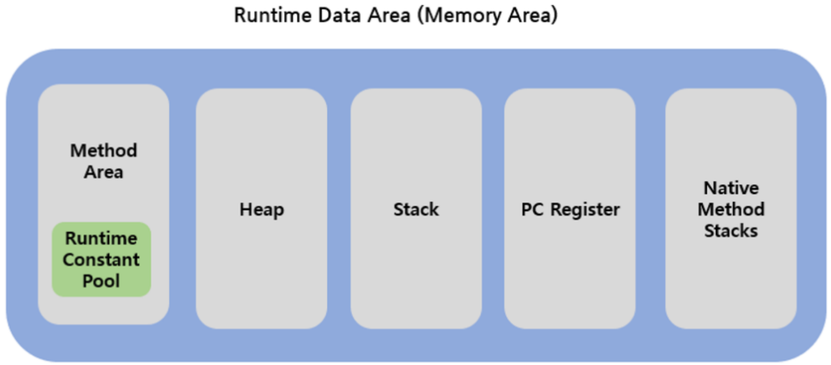
- 컴퓨터의 멀티태스킹(CPU 단일 코어 가정)
  - 스레드 A, B 존재
  - 운영체제는 하나의 스레드만 계속 실행하지 않고, 스레드 A, B를 각각 번갈아가며 실행
  - 그 과정에서 스레드가 어디까지 수행되었는지 그 위치를 찾아야 함
  - 그 위치를 찾아 계산하던 변수들의 값을 다시 CPU로 넣어야 하고 특정 스레드를 멈추는 시점에 CPU에서 사용하던 값들(Context, 문맥)을 메모리에 저장해 두어야 함, 그리고 이후에 그 스레드를 다시 실행할 때 다시 CPU로 불러들어와야 함 -> 컨텍스트 스위칭
- 멀티스레드는 대부분 효율적이지만 컨텍스트 스위칭 과정이 발생하므로 항상 효율적이라고 할 수 없음
- CPU 코어가 2개라면 스레드를 2개로 나눠 병렬 처리하는 것이 효율적, 모든 CPU를 사용하므로 연산을 2배 빠르게 처리 가능
- CPU 코어가 1개라면 스레드가 많아질수록 컨텍스트 스위칭을 처리하는 비용이 커지게 됨, 수행하는 것에 따라 단일 스레드로 처리하는 것이 나을 수도

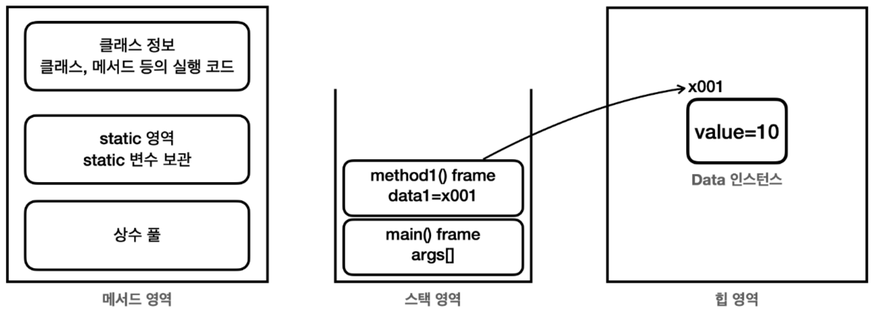
CPU 바운드 작업과 I/O 바운드 작업

- CPU 바운드 작업
  - CPU의 연산 능력을 많이 요구하는 작업 - 수학 계산, 데이터 분석/처리, 알고리즘 실행(인코딩) 등
  - CPU의 처리 속도가 작업 완료 시간을 결정하는 경우
- I/O 바운드 작업
  - 디스크, 네트워크, 파일 시스템 등 같은 입출력 작업을 많이 요구하는 작업
  - 입출력 작업이 완료될 때까지 대기 시간이 많이 발생하게 되어 그동안 CPU는 아이들링한 상태에 놓이게 됨 -> CPU를 사용하지 않고 대기상태로 두게 됨
  - ex) 데이터베이스 쿼리 처리, 파일 읽기/쓰기, 네트워크 통신, 사용자 입력 처리 등
- 웹 애플리케이션 서버 - I/O 작업 처리가 월등히 높음 -> 데이터베이스 호출하고 결과 처리 및 사용자의 입력을 기다리는 작업이 많기 때문
  - 일반적인 자바 웹 애플리케이션 경우, 사용자 요청 하나 처리하는데 스레드 1개 필요
- CPU 작업이 많은지, IO 작업이 많은지에 따라 스레드 숫자를 결정
  - CPU 작업: CPU 코어 수 + 1개
  - I/O 작업: CPU 코어 수 보다 많은 스레드 생성, CPU를 최대한 사용할 수 있는 숫자까지 스레드 생성
    - CPU를 많이 사용하지 않으므로 성능 테스트를 통해 CPU를 최대한 활용할 수 있는 **적절한** 숫자까지 스레드를 생성

2. 스레드 생성과 실행

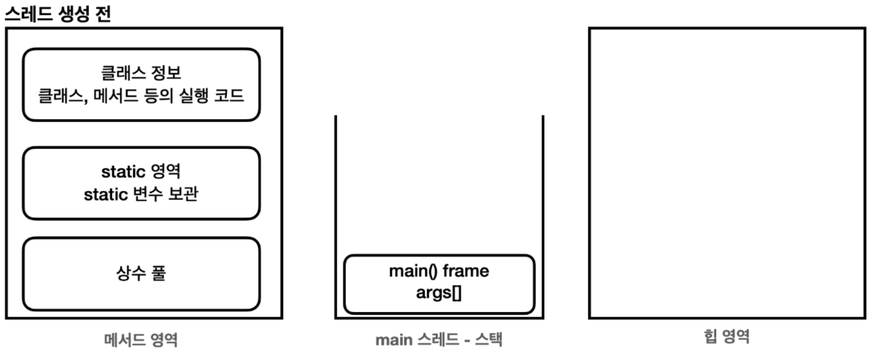
자바의 메모리 구조

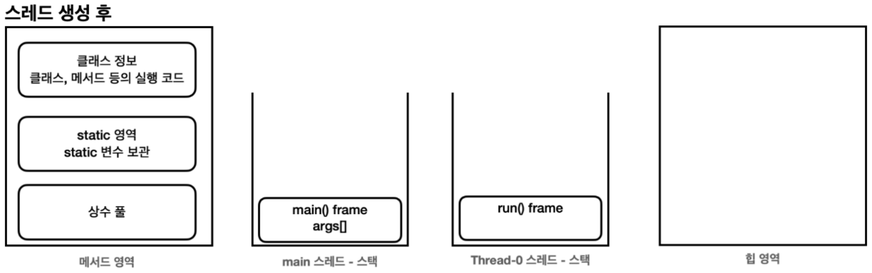




- 메서드 영역: 프로그램을 실행하는데 필요한 공통 데이터 관리, 프로그램의 모든 영역에서 공유됨
  - 클래스 정보: 클래스의 실행 코드(byte code), 필드, 메서드와 생성자 코드 등 모든 실행 코드가 존재
  - static 영역: static 변수 보관
  - 런타임 상수 풀: 프로그램을 실행하는데 필요한 공통 리터럴 상수 보관
- 스택 영역: 자바 실행 시 하나의 실행 스택이 생성됨, 각 스택프레임은 지역 변수, 중간 연산 결과, 메서드 호출 정보 등을 포함
  - 스택 프레임: 스택 영역에 쌓이는 것, 메서드를 호출할 때마다 하나의 스택 프레임이 쌓이고 메서드가 종료되면 해당 스택프레임은 제거
- 힙 영역: 객체(인스턴스)와 배열이 생성되는 영역, GC가 이뤄지는 주요 영역이고 더 이상 참조되지 않는 객체는 GC에 의해 제거

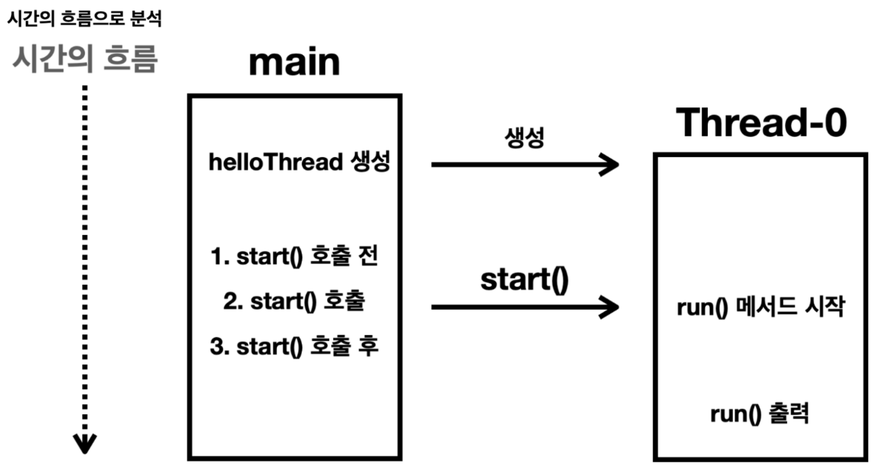
스레드 생성





- 스레드 생성 전
  - 프로세스가 실행되려면 적어도 스레드 하나는 있어야 함 -> main() thread
    메인 스레드는 프로그램 시작점인 main() 메서드를 실행
- 스레드 생성 후
  - HelloThread 스레드 객체를 생성하고 start() 메서드 호출 -> 자바는 새로 생성된 스레드만을 위한 별도의 스택 영역을 할당 **(반드시 객체 생성후 start()를 호출해야 스택 공간을 할당 받고 스레드가 작동함!)**
  - 스레드의 이름을 부여하지 않았다면, Thread-0, Thread-1과 같은 임의의 이름 부
  - Thread-0 스레드는 run() 메서드의 스택 프레임을 스택에 추가하면서 run() 메서드를 시작
- 메서드를 실행하면 스택 위에 스택 프레임이 쌓인다
  - main 스레드는 main() 메서드의 스택 프레임을 스택에 올리면서 시작
  - 직접 만든 스레드는 run() 메서드의 스택 프레임을 스택에 올리며 시작

스레드 실행 결과 (시간의 흐름으로 정리)



- main 스레드가 HelloThread 인스턴스 생성
- start() 메서드 호출 - 스레드가 시작되면서 스레드가 run() 메서드 호출
  - main 스레드가 run()을 실행하는 것이 아닌 생성된 스레드가 run()을 실행
  - main 스레드는 단지 start()를 통해 해당 스레드에게 실행을 지시하고 바로 start() 메서드를 빠져나옴
- main 스레드와 생성된 스레드는 동시에 실행됨
- main 스레드는 그림 1, 2, 3번 코드를 멈추지 않고 계속 수행, run() 메서드는 생성된 별도의 스레드에서 실행됨
- **스레드 간의 실행 순서는 보장되지 않는다.**
  - 스레드는 동시에 실행되기 때문에 스레드 간의 실행 순서는 언제든지 달라질 수 있음
  - 또한 실행 기간도 보장되지 않음(먼저 끝날 수도 있고… 번갈아 가며 수행될 수 있고..)

스레드 시작 2

start() vs run()

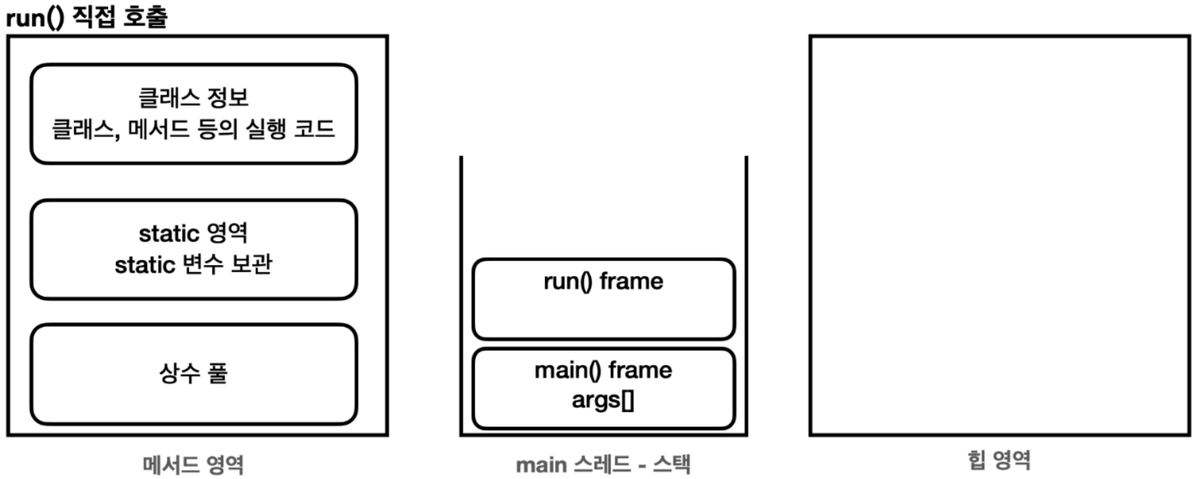
```
package thread.start;

public class BadThreadMain {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() start");

        HelloThread helloThread = new HelloThread();
        System.out.println(Thread.currentThread().getName() + ": run() 호출 전");
        helloThread.run(); // run() 직접 실행
        System.out.print                                    ": run() 호출 후");

        System.out.print                                    ": main() end");
    }

}
```

'run()' 호출을 'start()'로 바꿔야 할 것입니다
'start()'(으)로 바꾸기 ⌥⇧↵ 액션 더보기... ⌥↵
com.mardi2020.HelloThread
public void run()

start() 메서드를 호출해야 하지만,
의도한 run() 직접 호출한 예

```
> Task :Main.main()
main: main() start
main: run() call before
main: run()
main: run() call after
main: main() end
```

- start() 대신에 재정의한 run() 메서드를 직접 호출한다면?
  - 새로 생성한 스레드에서 run()을 실행하지 않음
  - Main 스레드에서 HelloThread 인스턴스의 run() 메서드를 직접 호출
  - Main 스레드에서 실행했으므로 main 스레드의 스택에 run()의 스택 프레임이 쌓임
- 결과적으로, main 스레드에서 모든 것을 처리한 셈 -> 생성한 스레드는 아무일도 하지 않음
- 스레드의 start() 메서드는 스레드에 스택 영역을 할당하는 아주 특별한 메서드 (필수적)
- 해당 스레드에서 재정의한 run() 메서드를 실행!!



데몬 스레드

User thread와 daemon thread

```java
public class DaemonThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run() start");
        try {
            Thread.sleep( millis: 10_000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(Thread.currentThread().getName() + ": run() end");
    }
}
```

```java
public class Main {
    신규 *
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() start");
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setDaemon(true);
        daemonThread.start();
        System.out.println(Thread.currentThread().getName() + ": main() end");
    }
}
```

실행결과:

```
> Task :Main.main()
main: main() start
main: main() end
Thread-0: run() start
```

- **사용자 스레드(non-daemon 스레드)**
  - 프로그램의 주요 작업 수행하고 작업이 완료될 때까지 실행됨
  - **모든 사용자 스레드가 종료되고 나서 JVM 종료**
- **데몬 스레드**
  - 사용자에게 직접적으로 보이지 않으면서 시스템 백그라운드에서 보조적인 작업 수행 (파일, 메모리 정리 작업 등)
  - 모든 사용자 스레드가 종료되면 데몬 스레드는 자동 종료
  - JVM은 데몬 스레드 종료여부와 상관없이(다른 모든 스레드가 종료될 때) 종료
  - setDaemon(true): 데몬스레드로 설정, start() 실행전 셋팅 필수이며 기본값은 false로 즉, 사용자 스레드가 기본
  - 데몬스레드 내에 “run( ) end” 출력전에 자바 프로그램 종료

데몬 스레드

User thread와 daemon thread

```java
public class DaemonThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run() start");
        try {
            Thread.sleep( millis: 10_000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println(Thread.currentThread().getName() + ": run() end");
    }
}
```

```java
public class Main {
    신규 *
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() start");
        DaemonThread daemonThread = new DaemonThread();
        daemonThread.setDaemon(false);
        daemonThread.start();
        System.out.println(Thread.currentThread().getName() + ": main() end");
    }
}
```

실행결과:

```
> Task :Main.main()
main: main() start
main: main() end
Thread-0: run() start
Thread-0: run() end
```

- setDaemon(false)로 설정해본 경우
  - Thread-0은 사용자 스레드로 설정
  - 즉, 메인 스레드가 종료되어도 사용자 스레드인 Thread-0이 작업을 마치고 종료될 때까지 자바 프로그램은 종료되지 않음
  - 따라서, 실행 결과에 “Thread-0 run() end”가 출력됨

스레드 생성

Runnable

```
package java.lang;

public interface Runnable {
    void run();
}
```

```
public class HelloThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": run()");
    }
}
```

```
public class Main {
    신규 *
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName() + ": main() start");
        HelloRunnable runnable = new HelloRunnable();
        Thread thread = new Thread(runnable);
        thread.start();
        System.out.println(Thread.currentThread().getName() + ": main() end");
    }
}
```

실행결과:

```
> Task :Main.main()
main: main() start
main: main() end
Thread-0: run()
```

- 앞의 경우는 Thread 클래스를 상속받아 스레드를 생성하는 방법
- **이번은 Runnable 함수형 인터페이스로 스레드를 구현하는 방법**
  - 앞 방법과 차이: **스레드와 해당 스레드가 실행할 작업이 서로 분리되어 있음**
  - 스레드 객체를 생성할 때 실행할 작업을 생성자로 전달함

Thread 상속과 Runnable 구현

- 스레드를 사용할 때는 Runnable 인터페이스를 구현하자
  - 스레드와 실행할 **작업이 명확히 분리됨**
  - 인터페이스를 사용하므로 클래스 상속이 가능
  - 유연하고 유지보수 하기 쉬운 코드 작성 가능??

	Thread (Class)	**Runnable (Interface)**
장점	간단한 구현: Thread 클래스를 상속받고 run() 메서드 재정의	상속의 자유로움: 다른 클래스를 상속받을 수 있음 코드의 분리: 스레드와 실행할 작업을 분리하여 코드 가독성 향상 여러 스레드가 동일한 Runnable 객체를 공유하여 자원 관리 효율화
단점	상속의 제한: 단일 상속만 가능 유연성 부족: 인터페이스를 사용하는 방법에 비해 유연성이 뒤떨어짐	코드가 약간 복잡해짐 Runnable 객체를 만들고 Thread에 전달하는 코드 추가

3. 스레드 제어와 생명 주기1

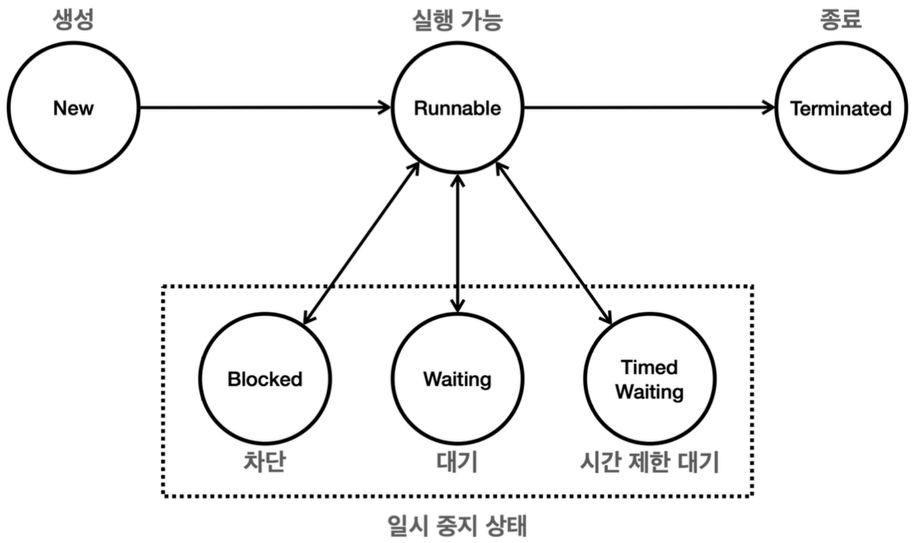
Runnable을 만드는 다양한 방법

실행 결과

```
//main 스레드 출력
09:55:58.709 [     main] mainThread = Thread[#1,main,5,main]
09:55:58.713 [     main] mainThread.threadId() = 1
09:55:58.713 [     main] mainThread.getName() = main
09:55:58.716 [     main] mainThread.getPriority() = 5
09:55:58.716 [     main] mainThread.getThreadGroup() =
java.lang.ThreadGroup[name=main,maxpri=10]
09:55:58.716 [     main] mainThread.getState() = RUNNABLE

//myThread 출력
09:55:58.717 [     main] myThread = Thread[#21,myThread,5,main]
09:55:58.717 [     main] myThread.threadId() = 21
09:55:58.717 [     main] myThread.getName() = myThread
09:55:58.717 [     main] myThread.getPriority() = 5
09:55:58.717 [     main] myThread.getThreadGroup() =
java.lang.ThreadGroup[name=main,maxpri=10]
09:55:58.717 [     main] myThread.getState() = NEW
```

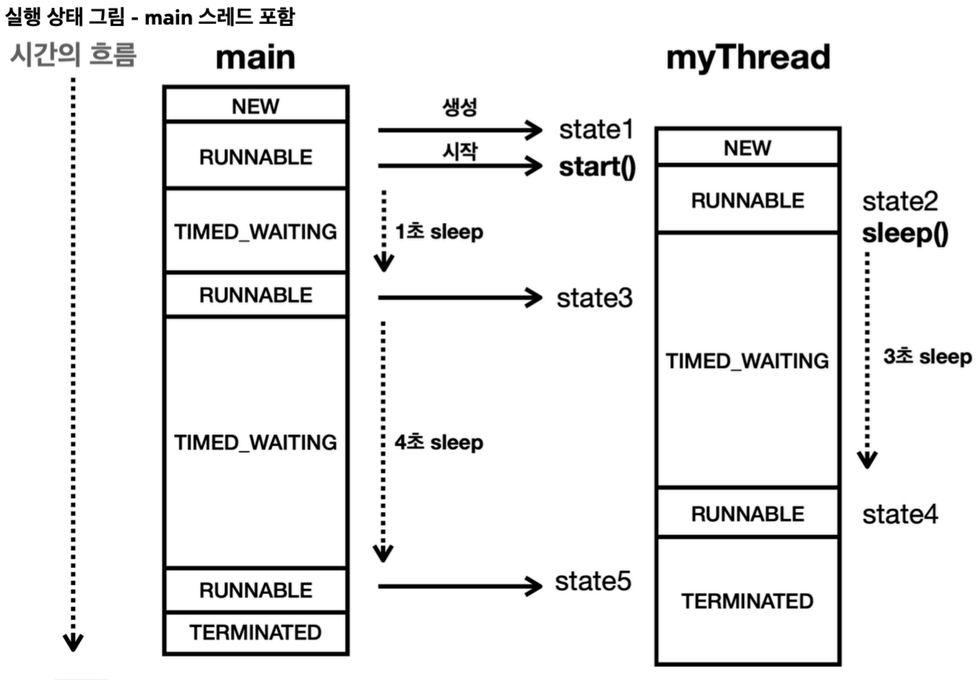
- Thread 클래스: 스레드 생성 + 관리 기능 제공
  - 스레드 생성: Runnable 인터페이스 구현체와 이름 전달
  - 스레드 객체 정보: `log("myThread = " + myThread);`
    스레드 id, 스레드 이름, 우선순위, 스레드 그룹 출력
    `Thread myThread = new Thread(new HelloRunnable(), "myThread");`
  - 스레드 id(threadId()): 스레드의 고유 식별자 반환(JVM 내에서 유일, 생성시 자동 할당되고 직접 지정 불가능)
  - 스레드 이름(getName()): 스레드 이름은 중복 가능
  - 스레드 우선순위(getPriority()): 우선순위 1(가장낮음) ~ 10(가장높음)까지의 값 설정 가능, 기본값은 5이며 setPriority() 메서드를 사용해 우선순위 지정 가능
    어떤 스레드의 실행을 결정하는데 관여하나 JVM 구현과 OS에 따라 실제 실행 순서에 따라 달라짐
  - 스레드 그룹(getThreadGroup()): 스레드가 속한 스레드 그룹 반환, 기본적으로 모든 스레드는 부모 스레드와 동일한 스레드 그룹에 속함
    하나의 그룹으로 묶어 특정 작업(일괄종료, 우선순위 설정 등) 편리하게 수행 가능
    부모 스레드: 새로운 스레드를 생성하는 스레드 *스레드 그룹은 잘 사용안함!
  - 스레드 상태(getState()): Thread.State 열거형 중 하나
    NEW / RUNNABLE / BLOCKED / WAITING / TIMED_WAITING / TERMINATE

스레드의 생명 주기



- **스레드의 상태**
  - **New**(새로운 상태): 스레드가 생성되었으나 시작되지 않음
  - **Runnable**(실행가능상태): 스레드가 실행중이거나 실행 준비가 된 상태
  - Suspended States -> 자바 스레드에서 Suspended states이라는 상태는 없고 설명하기 위한 용어임
    - **Blocked**(차단 상태): 스레드가 동기화 락을 기다리는 상태
    - **Waiting**(대기 상태): 스레드가 무기한으로 다른 스레드의 작업을 기다리는 상태
    - **Timed Waiting**(시간 제한 대기 상태): 스레드가 일정 시간동안 다른 스레드의 작업을 기다리는 상태
  - **Terminated**(종료 상태): 스레드의 실행이 종료된 상태

스레드의 생명 주기



- New: Thread 객체는 생성되었지만 start() 메서드가 호출되지 않은 상태
  Thread thread = new Thread(runnable);
- Runnable: start() 메서드가 호출되면 스레드는 이 상태로 진입
  thread.start();
- Block: 스레드가 다른 스레드에 의해 동기화 락을 얻기 위해 기다리는 상태
  synchronized (lock) { … } 코드 블록에 진입하려고 할 때, 다른 스레드가 이미 lock을 갖고 있어 기다리는 상태
- Waiting: 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한 기다리는 상태
  wait(), join() 메서드가 호출될 때 이 상태로 변경됨
  스레드는 다른 스레드가 notify(), notifyAll() 메서드를 호출하거나 join()이 완료될 때까지 기다림
  object.wait();
- Timed Waiting: 주어진 시간이 경과하거나 다른 스레드가 해당 스레드를 깨우면 상태에서 해제
  sleep(long mills), wait(long timeout), join(long miles) 메서드가 호출될 때 이상태가 됨
  Thread.sleep(1000);
- Terminated: 스레드가 정상적으로 종료되거나 예외가 발생해 종료된 경우 이상태로 돌아감, 스레드는 한번종료되면 다시 시작 불가

체크 예외 재정의

```java
package thread.control;

public class CheckedExceptionMain {

    public static void main(String[] args) throws Exception {
        throw new Exception();
    }

    static class CheckedRunnable implements Runnable {

        @Override
        public void run() /*throws Exception*/ { // 주석 풀면 예외 발생
            //throw new Exception(); // 주석 풀면 예외 발생
        }
    }
}
```

```java
static class MyRunnable implements Runnable {
    public void run() throws InterruptedException {
        Thread.sleep(3000);
    }
}
```

앞서 설명한 것처럼, sleep의 InterruptedException은 체크 예외여서 던질 수 없음 해당 코드는 컴파일 오류 발생

- Runnable 인터페이스의 run() 메서드를 구현할 때 InterruptedException 체크 예외를 밖으로 던질 수 없는 이유
  - **자바에서 메서드를 재정의할 때 지켜야할 예외와 관련된 규칙**
    - **체크 예외**
      - 부모 메서드가 체크 예외를 던지지 않는 경우, 재정의된 자식 메서드도 체크 예외를 던질 수 없음
      - 자식 메서드는 부모 메서드가 던질 수 있는 체크 예외의 하위 타입만 던짐
    - **언체크 예외** - 예외 처리를 강제하지 않으므로 상관없이 던질 수 있음
  - Runnable 인터페이스의 run() 메서드는 아무런 체크 예외를 던지지 않아 run() 메서드를 재정의하는 곳에서 체크 예외를 밖으로 던질 수 없음
  - **이런 제약이 있는 이유?**
    - 부모 클래스의 메서드를 호출하는 클라이언트 코드는 부모 메서드가 던지는 특정 예외만 처리하도록 작성됨
    - 자식 클래스가 더 넓은 범위의 예외를 던지게 되면, 모든 예외를 제대로 처리 못할 수 있음
    - 이는 **예외 처리 일관성을 해치고** 예상치 못한 Runtime 오류를 발생시킬 위험 증가
    - Parent p = new Child(); p.method(); 를 호출했을때
      Parent p는 InterruptedException을 반환하는데 그 child가 전혀 다른 예외를 반환한다면 클라이언트는 해당 예외를 잡을 수 없게 됨

join - 시작

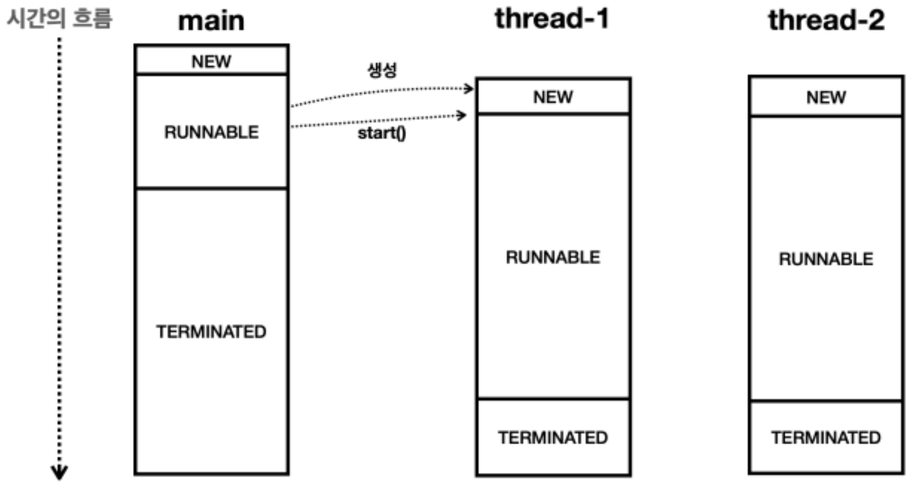
```java
public class JoinMainV0 {

    public static void main(String[] args) {
        log( obj: "Start");
        Thread thread1 = new Thread(new Job(), name: "thread-1");
        Thread thread2 = new Thread(new Job(), name: "thread-2");

        thread1.start();
        thread2.start();
        log( obj: "End");
    }

    static class Job implements Runnable {  2 usages

        @Override
        public void run() {
            log( obj: "작업 시작");
            sleep( millis: 2000);
            log( obj: "작업 완료");
        }
    }
}
```

- **join( ) 메서드를 통해 WAITING(대기 상태)를 확인**
- Waiting (대기 상태): 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한 기다리는 상태

실행 결과:

```
09:07:08.545 [     main] Start
09:07:08.545 [     main] End
09:07:08.545 [ thread-2] 작업 시작
09:07:08.545 [ thread-1] 작업 시작
09:07:10.561 [ thread-2] 작업 완료
09:07:10.561 [ thread-1] 작업 완료
```

join - 시작



그림에는 생략 되었지만 thread-2도 메인 스레드가 생성하고 start() 메서드를 호출해서 스택 공간을 할당받음!

또한 RUNNABLE 상태로 되어있지만, 실제로는 TIMED_WAITING 상태

- 2초가 걸리는 작업(= sleep(2000)) 가정
- 실행결과, 메인 스레드 먼저 종료되고 thread-1, thread-2 가 종료됨
  - main 스레드는 thread-1, thread-2을 실행하고 바로 자신의 다음 코드 실행
  - **메인 스레드는 다른 스레드가 끝날때까지 기다리지 않음**
  - 메인 스레드는 그저 start( )를 호출해 다른 스레드를 실행시키고 바로 자신의 다음 코드를 실행하게 됨
- **this 키워드**
  - 인스턴스의 메서드를 호출하면 어떤 인스턴스의 메서드를 호출했는지 기억하기 위해 해당 인스턴스의 참조값을 스택 프레임 내부에 저장하는 것
  - this는 호출된 인스턴스 메서드가 소속된 객체를 가리키는 참조, 이것이 스택 프레임 내부에 저장되어 있음!

join - sleep 사용

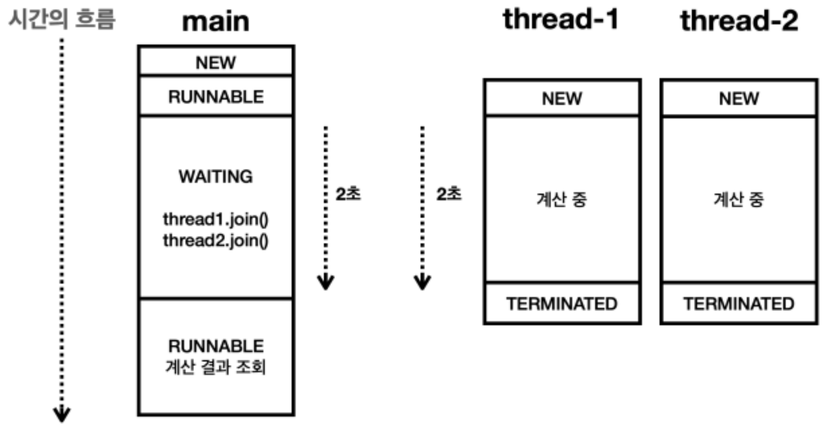
특정 스레드를 기다리게 하는 가장 간단한 방법

- main 스레드를 가장 마지막에 종료하고 싶다면?
  - ex) thread-1, thread-2 각 스레드의 작업 결과를 받아 main 스레드에서 처리하고 싶은 경우
- 하지만 sleep으로 무작정 기다리는 방법은 대기 시간에 손해도 보고, 기다려야 하는 스레드의 작업 완료 시간을 정확히 알기 어려움
- 더 나은 방법: 메인 스레드가 반복문을 사용해서 thread-1, thread-2의 상태가 TERMINATED가 될 때까지 계속 체크하는 방법
  - while(thread.getState() != TERMINATED) { ... }
  - 당연히, 이 방법도 번거롭고 무한정 반복되는 반복문은 CPU 연산을 계속 사용하게 됨
- **join( ) 메서드**를 사용하여 문제 해결

join - join 사용

특정 스레드가 완료될 때까지 기다려야 하는 상황에서 사용

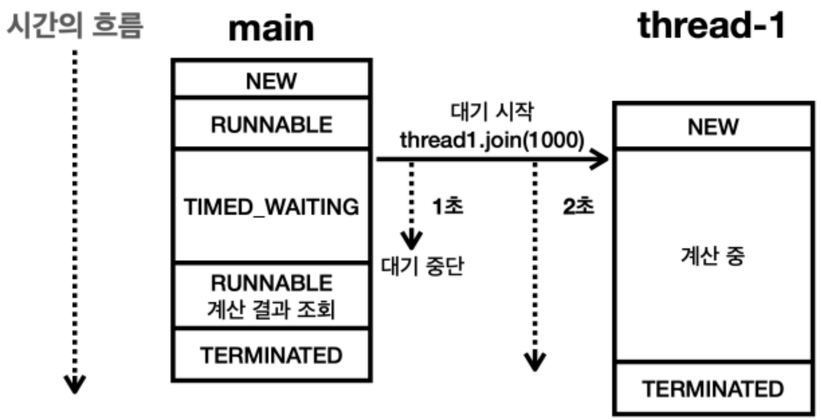
```
10:03:49.867 [     main] Start
10:03:49.867 [     main] join() - main 스레드가 thread1, thread2 종료까지 대기
10:03:49.867 [ thread-1] 작업 시작
10:03:49.867 [ thread-2] 작업 시작
10:03:51.893 [ thread-1] 작업 완료 result = 1275
10:03:51.893 [ thread-2] 작업 완료 result = 3775
10:03:51.893 [     main] main 스레드 대기 완료
10:03:51.893 [     main] task1.result = 1275
10:03:51.893 [     main] task2.result = 3775
10:03:51.893 [     main] task1 + task2 = 5050
10:03:51.893 [     main] End
```



- thread.start( ) 후 **thread.join( );** 메서드 추가하여 메인 스레드를 대기하도록 함
  - main 스레드는 thread-1, thread-2가 종료될 때까지 WAITING 상태로 변경됨
- 참고) join( ) 메서드는 InterruptedException을 던짐
- Waiting (대기 상태)
  - 스레드가 다른 스레드의 특정 작업이 완료되기를 무기한으로 기다리는 상태
  - join( )을 호출하는 스레드는 대상 스레드가 TERMINATED 상태가 될 때까지 대기
  - TERMINATED 상태가 되면 호출 스레드는 다시 RUNNABLE 상태가 되며 다음 코드 수행
- 단점!
  - 다른 스레드가 완료될 때까지 무기한으로 기다려야 함
  - **그래서, 다른 스레드의 작업을 일정 시간 동안만 기다리고 싶다면...?**

join - 특정 시간 만큼만 대기

예제) 스레드 1개, 작업도 1개




```
10:12:12.995 [     main] Start
10:12:12.995 [     main] join(1000) - main 스레드가 thread1 종료까지 1초 대기
10:12:12.995 [ thread-1] 작업 시작
10:12:14.013 [     main] main 스레드 대기 완료
10:12:14.013 [     main] task1.result = 0
10:12:14.013 [     main] End
10:12:15.009 [ thread-1] 작업 완료 result = 1275
```

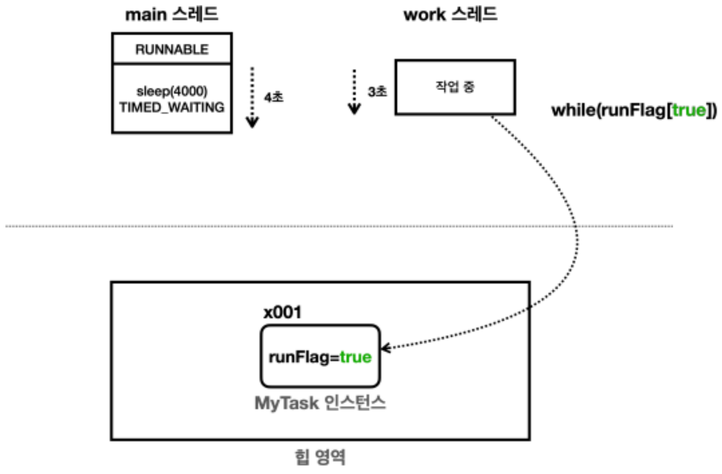
- join( ) 두가지 방법
  - join( ): 호출 스레드는 대상 스레드가 완료될 때까지 무한정 대기
  - join(ms): 호출 스레드는 특정 시간(ms)만큼 대기, 호출 스레드는 지정 시간이 지나면 다시 RUNNABLE 상태로 변하고 다음 코드 수행
- main 스레드는 join(1000)을 통해 thread-1을 1초 대기를 하게 됨
  - 메인 스레드는 **TIMED**_WAITING 상태가 됨 (특정 시간만큼 대기하므로 WAITING 상태가 아님)
  - thread-1의 작업에 2초가 소요
  - 따라서 1초가 지나도 thread-1의 작업이 완료되지 않아 메인 스레드는 대기 중단하고 RUNNABLE 상태로 바뀐뒤 다음 코드를 수행
    - 이때 thread-1의 작업이 아직 완료되지 않아 task1.result = 0이 출력
  - 메인 스레드가 종료된 이후 thread-1이 계산을 끝냄, 따라서 result = 1275 출력

정리) 다른 스레드가 끝날 때까지 무한정 기다려야 한다면 join() , 다른 스레드의 작업을 무한정으로 기다릴 수 없다면 join(ms) 사용

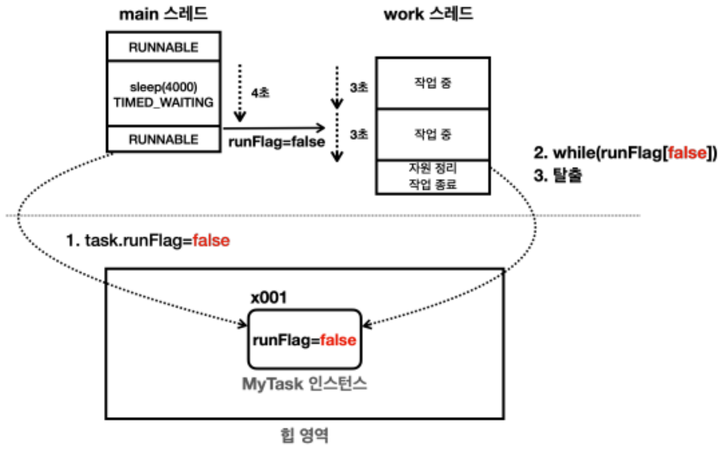
4. 스레드 제어와 생명 주기2

인터럽트 - 시작1

특정 스레드의 작업을 중간에 중단



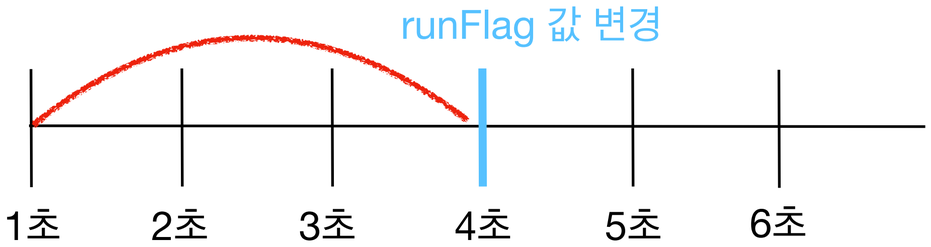
- `work` 스레드는 `runFlag`가 `true`인 동안 계속 실행된다.



- 스레드 작업을 중단하는 가장 쉬운 방법: 변수(runFlag를 사용해 스레드에 작업 중단 지시)
- but, 메인 스레드가 runFlag=false를 통해 작업 중단을 지시하더라도 work 스레드가 바로 반응하지 않고 2초 정도 지나고 나서 저원 정리 후 작업 종료
- 아래 코드의 sleep(3000) 때문

```
while (runFlag) {
    log("작업 중");
    sleep(3000);
}
```

- 메인 스레드가 runFlag를 변경해도 work 스레드는 sleep(3000)를 통해 3초간 잠들어 있음, 3초 뒤에 while(runFlag) 를 실행해야 runFlag를 확인하고 작업 중단 가능
- runFlag 변경후 2초 후에 작업 종료되는 이유: work 스레드가 3초에 한번씩 깨어나서 runFlag 값을 확인하는데 메인 스레드가 4초에 runFlag를 변경했기 때문



인터럽트 - 시작2

WAITING, TIMED_WAITING 상태를 직접 깨운다

```
10:56:12.261 [     work] 작업 중
10:56:15.267 [     work] 작업 중
10:56:16.254 [     main] 작업 중단 지시 thread.interrupt()
10:56:16.263 [     main] work 스레드 인터럽트 상태1 = true
10:56:16.263 [     work] work 스레드 인터럽트 상태2 = false
10:56:16.263 [     work] interrupt message=sleep interrupted
10:56:16.263 [     work] state=RUNNABLE
10:56:16.263 [     work] 자원 정리
10:56:16.263 [     work] 자원 종료
```

- thread.interrupt( ) : 해당 스레드에 인터럽트 발생
  - 인터럽트가 발생하면 해당 스레드에 InterruptedException 발생 하지만, 즉각 발생하지 않음
    - 즉, while(true), log("작업 중") 라인에서는 해당 예외가 발생하지 않음
    - Thread.sleep() 처럼 예외를 던지는 메서드를 호출하거나 또는 호출하며 대기중일 때 예외 발생
  - 인터럽트를 받은 스레드는 대기 상태 -> RUNNABLE 상태 전환하여 코드 수행
  - 이때, 스레드가 RUNNABLE 상태이므로 예외를 catch로 잡아 정상 흐름을 변경하면 됨
- 인터럽트가 적용되고 나서 예외가 발생하면 해당 스레드는 RUNNABLE 상태가 되고 인터럽트 발생 상태도 정상으로 돌아오게 됨
- 인터럽트를 사용하면 대기중인 스레드를 바로 깨워 실행 가능한 상태로 변경가능
- 앞의 runFlag 변수 사용 방식보다 **즉각 반응 하여 적용**할 수 있음!

인터럽트 - 시작3

```
while (true) {  //인터럽트 체크 안함
    log("작업 중");
    Thread.sleep(3000); //여기서만 인터럽트 발생
}
```

while(true) 부분은 체크하지 않음 -> 인터럽트가 발생하더라도 이 부분은 항상 참이여서 다음 코드를 넘어가게 되고, sleep( ) 호출 뒤 인터럽트가 발생하게 됨

```
while (인터럽트_상태_확인) {  //여기서도 인터럽트 상태 체크
    log("작업 중");
    Thread.sleep(3000); //인터럽트 발생
}
```

while 문을 체크하는 부분에서 확인할 수 있게 되어 빠르게 빠져나갈 수 있음

```
while (인터럽트_상태_확인) {  //여기서도 체크
    log("작업 중");
}
```

인터럽트의 상태를 직접 확인하면, 인터럽트를 발생 시키는 sleep( )과 같은 코드가 없어됨

인터럽트 - 시작3

```
static class MyTask implements Runnable {

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) { // 인터럽트 상태 변경X
            log( obj: "작업 중");
        }
        log( obj: "work 스레드 인터럽트 상태2 = " + Thread.currentThread().isInterrupted());

        try {
            log( obj: "자원 정리");
            Thread.sleep( millis: 1000);
            log( obj: "자원 종료");
        } catch (InterruptedException e) {
            log( obj: "자원 정리 실패 - 자원 정리 중 인터럽트 발생");
            log( obj: "work 스레드 인터럽트 상태3 = " + Thread.currentThread().isInterrupted());
        }
        log( obj: "작업 종료");
    }
}
```

Thread.currentThread().isInterrupted()
현재 실행되는 스레드가 인터럽트 상태인지 조회

```
15:14:24.142 [     work] 작업 중
15:14:24.142 [     work] 작업 중
15:14:24.142 [     work] 작업 중
15:14:24.142 [     main] 작업 중단 지시 thread.interrupt()
15:14:24.142 [     work] 작업 중
15:14:24.150 [     work] work 스레드 인터럽트 상태2 = true
15:14:24.150 [     work] 자원 정리
15:14:24.150 [     work] 자원 정리 실패 - 자원 정리 중 인터럽트 발생
15:14:24.150 [     main] work 스레드 인터럽트 상태1 = true
15:14:24.150 [     work] work 스레드 인터럽트 상태3 = false
15:14:24.150 [     work] 작업 종료
```

- 메인스레드는 interrupt( ) 메서드를 사용해서 work 스레드에 인터럽트를 검
- work 스레드는 인터럽트 상태로, isInterrupted( ) = true
- 이때, while 조건이 false로 변경되면서 탈출, *while (!true) -> while (false)*
- **이 코드의 문제**
  - work 스레드의 인터럽트 상태가 계속 유지되는 점
  - 인터럽트 예외가 발생했을 경우 스레드의 인터럽트 상태는 false
  - 반면, isInterrupted( ) 메서드는 인터럽트의 상태를 변경하지 않음 (단순히 스레드가 인터럽트 상태인지만 알려줌)
- work 스레드는 이후에 자원 정리 코드를 실행하는데 이때도 인터럽트의 상태는 계속 true로 유지됨
  이때 인터럽트가 발생하는 sleep( ) 같은 메서드를 수행한다면 해당 코드에서 인터럽트 예외가 발생
- 우리가 기대한 것: while문을 탈출하기 위해서 딱 한번만 인터럽트를 사용하는 것 뿐이지 다른 곳에서도 계속해서 인터럽트가 발생하는 것이 아님
- 자원 정리 도중, 인터럽트가 발생하게 되어 자원 정리를 실패하게 됨
- 자바에서 인터럽트 예외가 한번 발생하면 스레드의 인터럽트 상태를 다시 정상(false)로 돌리는 것은 이런 이유 때문
  스레드의 인터럽트 상태를 정상으로 돌리지 않으면 이후에도 계속 인터럽트가 발생됨
- 따라서 인터럽트의 목적 달성시 인터럽트 상태를 다시 정상으로 돌려두어야 함
  - while (인터럽트_상태_확인) 부분에서 인터럽트의 상태를 확인하고 true라면 **인터럽트 상태를 다시 false로 돌려두면 됨**

인터럽트 - 시작 4

Thread.interrupted( )

```
15:20:51.503 [     work] 작업 중
15:20:51.503 [     work] 작업 중
15:20:51.503 [     main] 작업 중단 지시 thread.interrupt()
15:20:51.503 [     work] 작업 중
15:20:51.508 [     main] work 스레드 인터럽트 상태1 = true
15:20:51.508 [     work] work 스레드 인터럽트 상태2 = false
15:20:51.508 [     work] 자원 정리
15:20:52.513 [     work] 자원 종료
15:20:52.514 [     work] 작업 종료
```

- isInterrupted( ): 스레드의 인터럽트 상태 단순 체크
- interrupted( ): 직접 체크해서 사용할 때
  - 스레드가 인터럽트 상태: return true, 해당 스레드의 인터럽트 상태를 false로 변경
  - 인터럽트 상태가 아니라면: return false, 인터럽트 상태 변경을 하지 않음
- 코드

```
while (!Thread.interrupted()) { //인터럽트 상태 변경O
    log("작업 중");
}
```

- while (!Thread.interrupted()) => while (!true) => while (false) 흐름으로 탈출
- 자원 정리시 인터럽트의 상태는 false이므로 인터럽트가 발생하는 sleep()과 같은 코드를 수행하더라도 인터럽트가 발생하지 않음, 자원 정리 정상적으로 마무리됨
- **자바는 인터럽트 예외가 한번 발생하게 되면 스레드의 인터럽트 상태를 다시 false로 돌려 놓음**

yield - 양보하기
sleep( )의 실행 결과

- 같은 코드를 sleep( )과 yield( )를 추가해 호출해 본 결과

```java
public class YieldMain {
    1 usage
    static final int THREAD_COUNT = 1000;

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread thread = new Thread(new MyRunnable());
            thread.start();
        }
    }

    1 usage
    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName() + " - " + i);
                sleep( millis: 1); // 2. sleep
            }
        }
    }
}
```

sleep(1)을 추가한 실행 결과

스레드의 상태를 1밀리초 동안 RUNNABLE -> TIMED_WAITING 변경
스레드는 CPU 자원을 사용하지 않고 **실행 스케줄링에서 잠시 동안 제외**

1밀리초의 대기 시간 이후 다시 TIMED_WAITING -> RUNNABLE 상태가 되면서
실행 스케줄링에 포함

TIMED_WAITING 상태가 되면서 다른 스레드에 실행 양보, 큐에 대기중인 다른
스레드가 CPU 실행 기회를 더 빨리 얻게 됨

BUT, (RUNNABLE -> TIMED_WAITING -> RUNNABLE) 로 변경되는 복잡한 과정을 거치고 특정
시간만큼 스레드가 실행되지 않음

양보할 스레드가 없다면…, 나머지 스레드가 모두 대기 상태로 있어도 내 스레드까지 잠깐 실행되지 않음
(즉, 양보할 사람이 없는데 혼자서 양보한 상황)

yield - 양보하기

yield( )의 실행 결과

- 같은 코드를 sleep( )과 yield( )를 추가해 호출해 본 결과

```java
public class YieldMain {
    1 usage
    static final int THREAD_COUNT = 1000;

    public static void main(String[] args) {
        for (int i = 0; i < THREAD_COUNT; i++) {
            Thread thread = new Thread(new MyRunnable());
            thread.start();
        }
    }

    1 usage
    static class MyRunnable implements Runnable {

        @Override
        public void run() {
            for (int i = 0; i < 10; i++) {
                System.out.println(Thread.currentThread().getName() + " - " + i);
                Thread.yield(); // 3. yield
            }
        }
    }
}
```

yield( )을 추가한 실행 결과

스레드가 RUNNABLE 상태라면 (자바에서는 두 상태를 분리할 수 없음, OS 상의 개념)
- **실행상태(running):** 스레드가 CPU에서 실행 중
- **실행 대기 상태(ready)**: CPU 처리를 큐에서 대기 중인 상태

yield( ) 작동 방식
- Thread.yield( ) : 현재 실행 중인 스레드가 자발적으로 CPU를 양보하여 다른 스레드가 실행 될 수 있도록 함
- yield( ) 메서드를 호출한 스레드는 **RUNNABLE 상태를 유지**하면서 CPU 양보(**이 스레드는 다시 큐에 들어가면서 다른 스레드에게 CPU 사용기회 전달**)

    - 양보할 스레드가 없다면 **해당 스레드가 계속해서 실행됨**을 의미함

yield( )는 OS의 스케줄러에게 힌트만 제공할 뿐, **강제적인 실행 순서를 지정하지 않음**