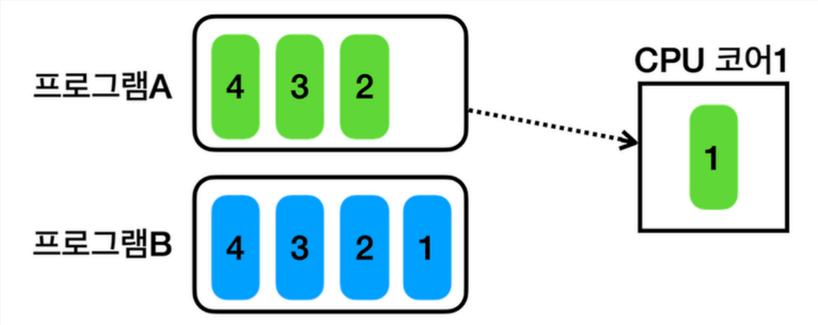
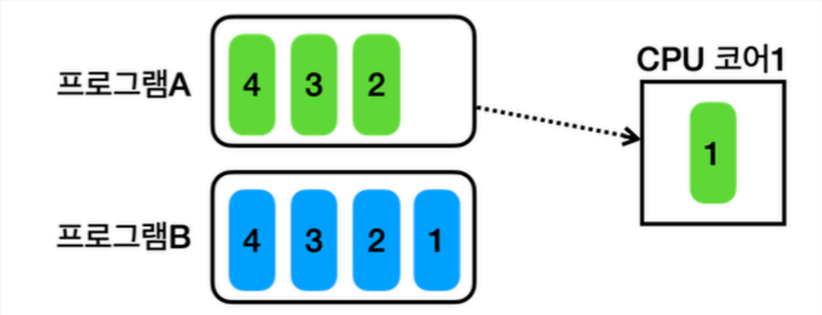
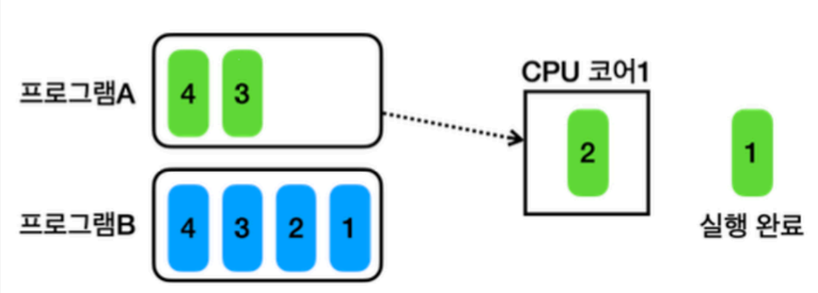
1주차 스터디

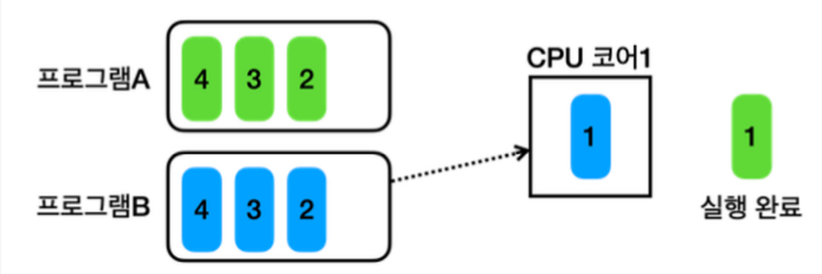
목차

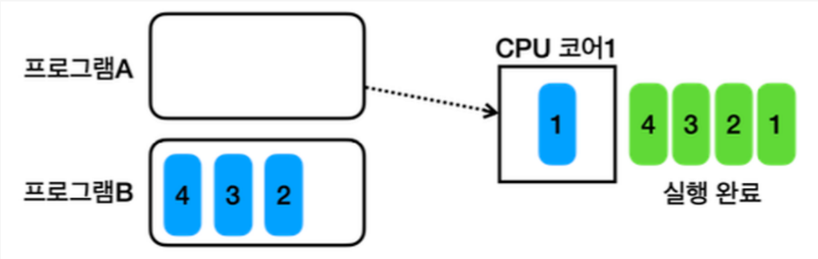
단일 프로그램 vs 멀티 태스킹

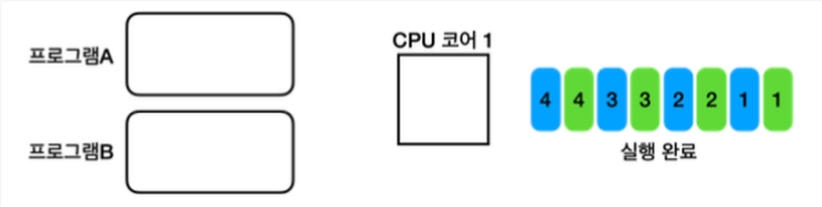












멀티 프로세싱 vs 멀티 태스킹

멀티프로세싱은 하드웨어 장비의 관점이고, 멀티태스킹은 운영체제 소프트웨어의 관점이다.

멀티 프로세싱

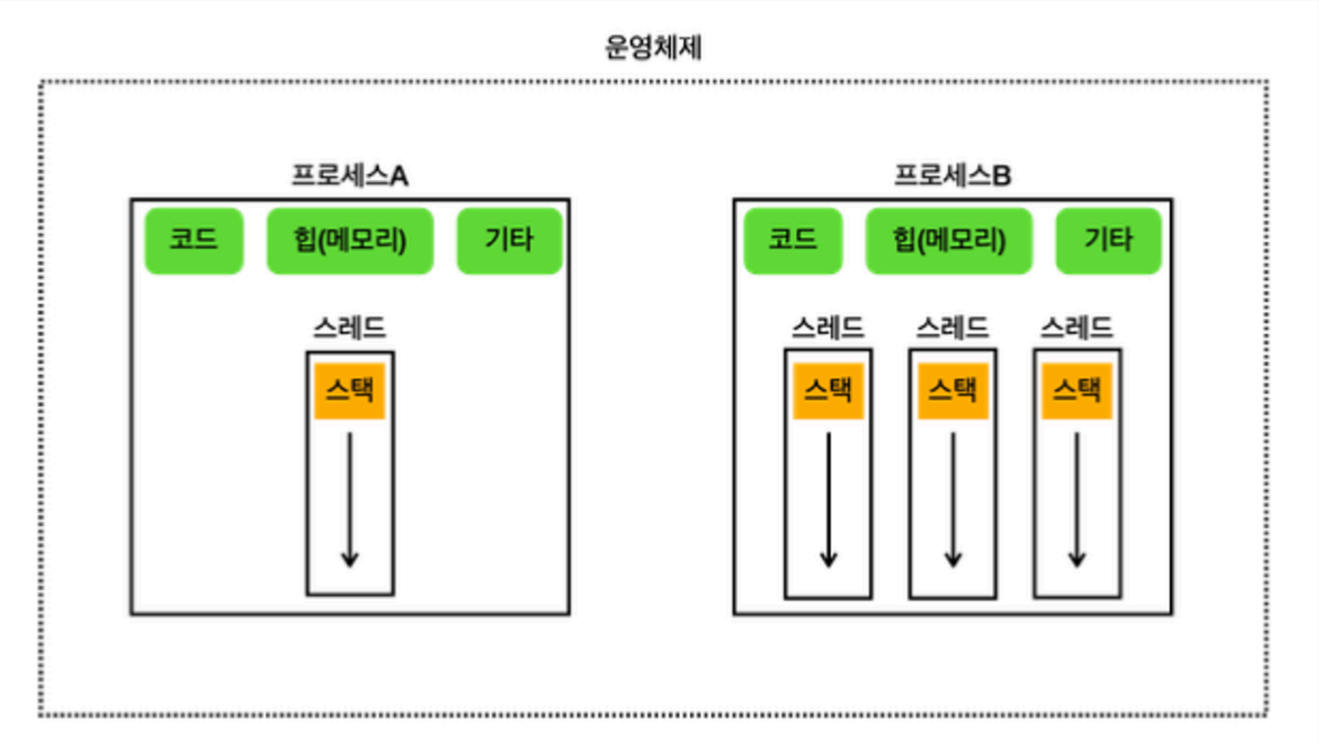
- 여러 CPU(여러 CPU 코어)를 사용하여 동시에 여러 작업을 수행하는 것을 의미
- 하드웨어 기반으로 성능을 향상시킨다.
- 예: 다중 코어 프로세서를 사용하는 현대 컴퓨터 시스템

멀티 태스킹

- 단일 CPU(단일 CPU 코어)가 여러 작업을 동시에 수행하는 것처럼 보이게 하는 것을 의미
- 소프트웨어 기반으로 CPU 시간을 분할하여 각 작업에 할당한다.
- 예: 현대 운영 체제에서 여러 애플리케이션이 동시에 실행되는 환경

프로세스

- 프로세스란 메인 메모리로 이동하여 실행 중인 프로그램을 의미한다.
- 각 프로세스는 독립적인 메모리 공간을 갖고 있다.
- 운영체제에서 별도의 작업 단위로 분리해서 관리된다.
- 각 프로세스는 별도의 메모리 공간을 갖고 있기 때문에 서로 간섭하지 않는다.



프로세스의 메모리 구성

- 코드 섹션: 실행할 프로그램의 코드가 저장되는 부분
- 데이터 섹션: 전역 변수 및 정적 변수가 저장되는 부분(그림에서 기타에 포함)
- 힙 (Heap): 동적으로 할당되는 메모리 영역
- 스택 (Stack): 메서드(함수) 호출 시 생성되는 지역 변수와 반환 주소가 저장되는 영역(스레드에 포함)
- 프로세스는 스레드를 위한 실행 환경과 자원을 제공하는 컨테이너 역할

스레드

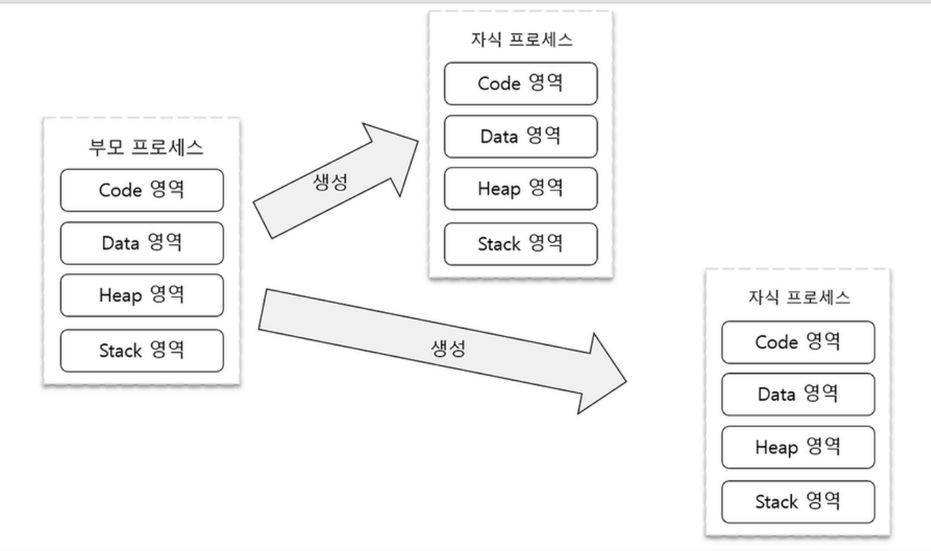
멀티 프로세스 운영체제 기반 프로그램의 문제점과 새로운 제안

- 두 가지 이상의 일을 동시에 처리하기 위해서, 혹은 둘 이상의 실행 흐름이 필요해서 추가적으로 프로세스를 생성하는 작업은 상당히 부담
- 많은 수의 프로세스 생성은 빈번한 컨텍스트 스위칭으로 이어져 성능에 영향
- 이러한 문제를 해결하기 위해 등장한 스레드

스레드

- 스레드는 하나의 프로그램 내에서 여러 개의 실행 흐름을 두기 위한 모델이다
- 스레드는 프로세스처럼 완벽히 독립적인 구조가 아니다. 스레드들 사이에는 공유하는 요소들이 있다.
- 스레드는 공유하는 요소가 있는 관계로 컨텍스트 스위칭에 걸리는 시간이 프로세스보다 짧다.

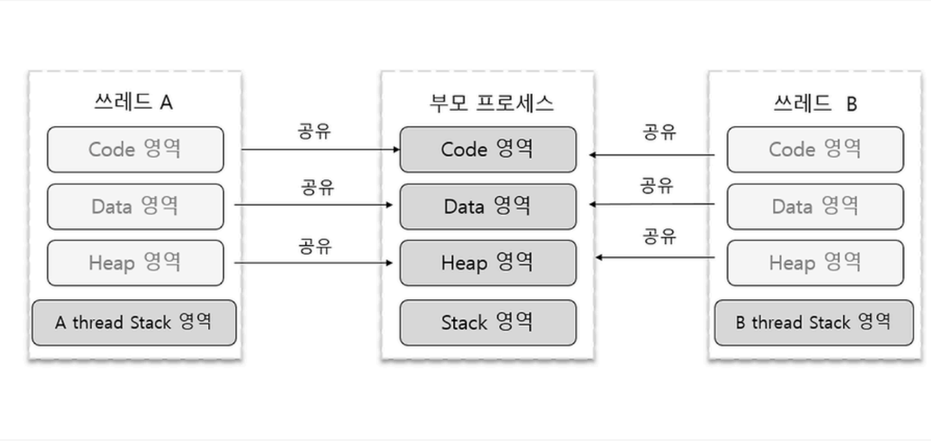
스레드



메모리 구조 관점에서 본 프로세스와 스레드

프로세스

- 자식 프로세스가 생성되고 난 다음에는 모든 것이 부모 프로세스와 독립적
- 프로세스 간에 데이터를 주고받기 위해서 IPC라는 메커니즘이 필요

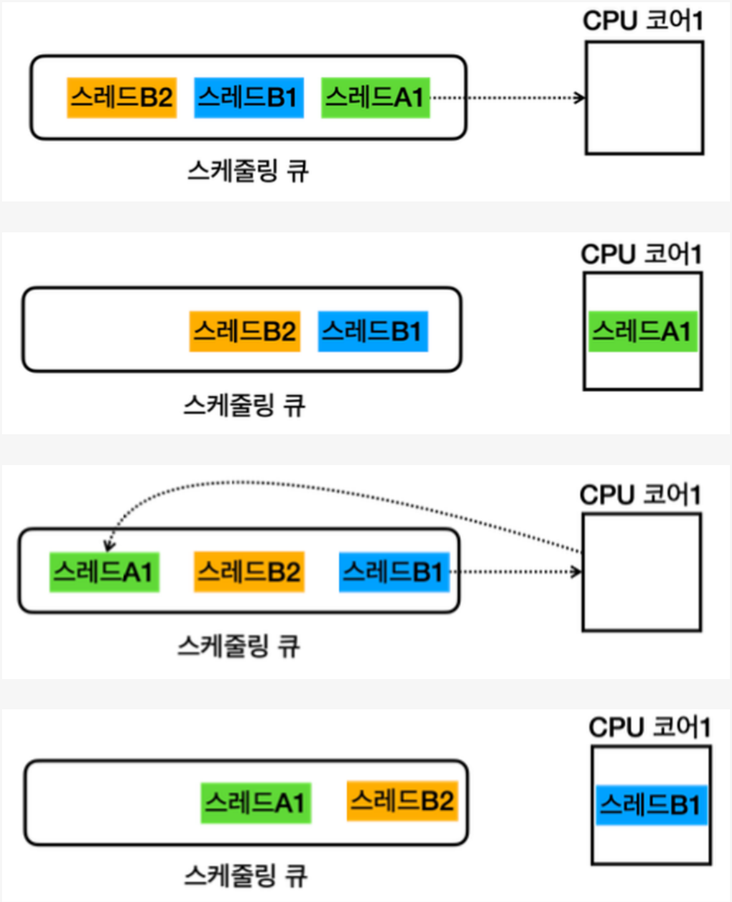


스레드

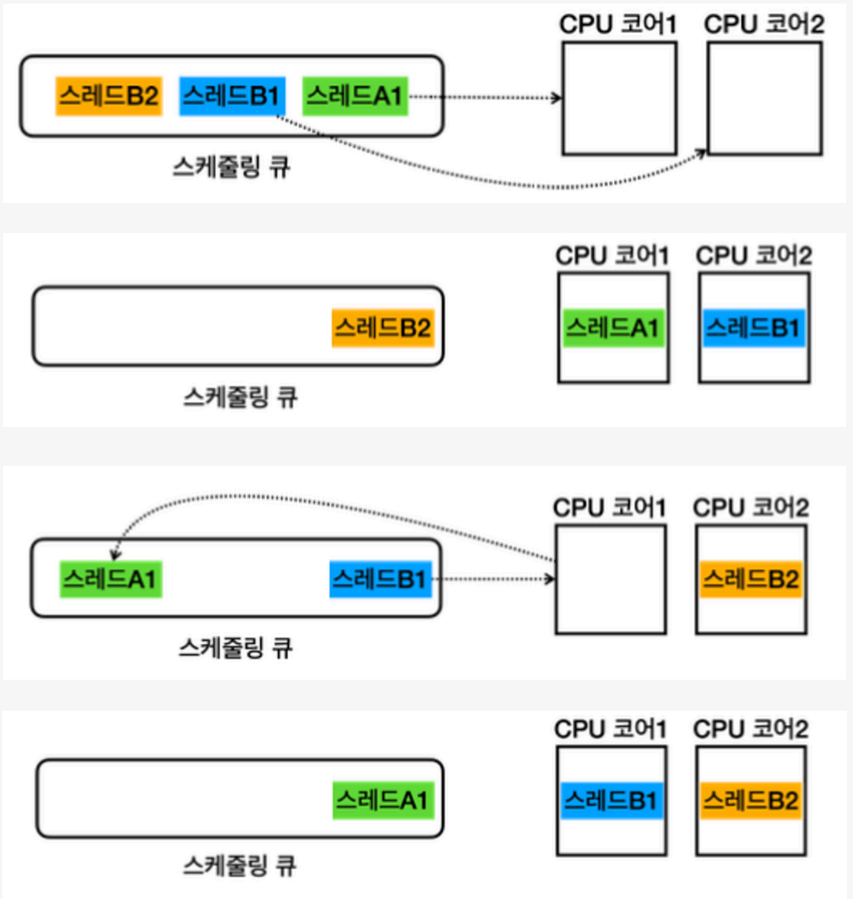
- 같은 프로세스의 코드 섹션, 데이터 섹션, 힙(메모리)은 프로세스 안의 모든 스레드가 공유
- 각 스레드는 자신만의 개별적인 스택을 갖고 있다. 다른 스택에 접근할 수 없다.

스레드와 스케줄링

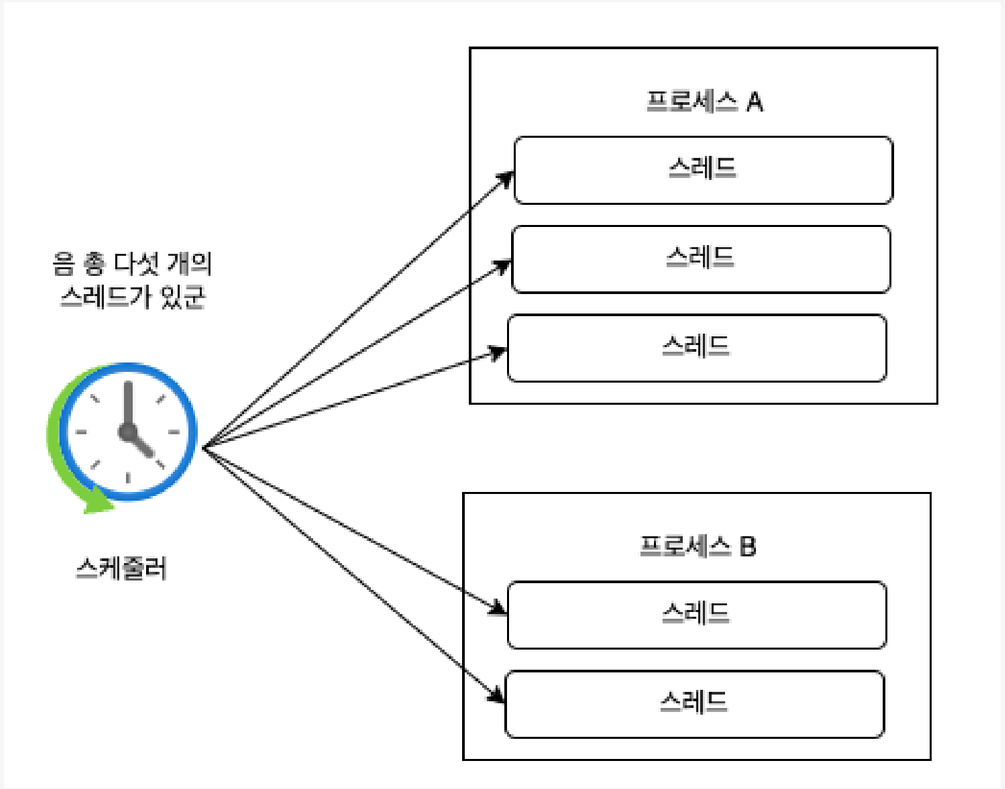
단일 코어 스케줄링



멀티 코어 스케줄링



컨텍스트 스위칭



스레드 컨텍스트 스위칭

- 그림을 보면 프로세스 B안에 두 개의 스레드가 존재한다.
- 이 둘은 하나의 프로세스 내에 존재하므로 별개의 스레드가 아니다.
- 이 둘 사이에서 발생하는 컨텍스트 스위칭을 스레드 컨텍스트 스위칭이라 한다.

프로세스 컨텍스트 스위칭

- 서로 다른 프로세스 내에 존재하는 스레드들 사이에서 발생하는 컨텍스트 스위칭을 의미
- 프로세스는 다른 프로세스와 공유하는 영역이 없다.
- 따라서 스레드 컨텍스트 스위칭에 비교해서 속도가 느리다.

스레드 생성

Thread 상속

```
public class StartTest1Main {

    public static void main(String[] args) {
        CounterThread thread = new CounterThread();
        thread.start();
    }

    static class CounterThread extends Thread {

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                log("value: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

Runnable 구현

```
public class StartTest2Main {

    public static void main(String[] args) {
        Thread thread = new Thread(new CounterRunnable(), "counter");
        thread.start();
    }

    static class CounterRunnable implements Runnable {

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                log("value: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    }
}
```

Thread 상속 vs Runnable 구현

Thread 클래스 상속 방식

장점

- 간단한 구현: Thread 클래스를 상속받아 run() 메서드만 재정의하면 된다.

단점

- 상속의 제한: 다른 클래스를 상속받고 있는 경우 Thread 클래스를 상속 받을 수 없다.
- 유연성 부족: 인터페이스를 사용하는 방법에 비해 유연성이 떨어진다.

Runnable 구현

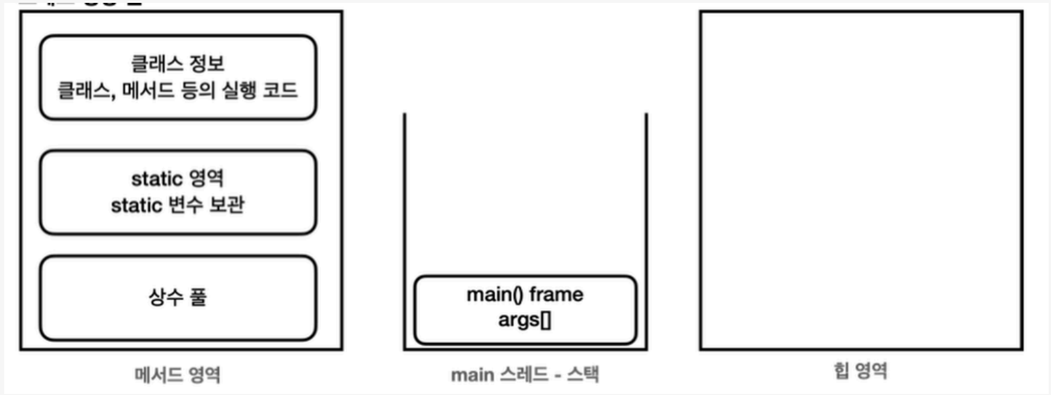
장점

- 상속의 자유로움: Runnable 인터페이스 방식은 다른 클래스를 상속받아도 문제없이 구현할 수 있다.
- 코드의 분리: 스레드와 실행할 작업을 분리하여 코드의 가독성을 높일 수 있다.
- 여러 스레드가 동일한 Runnable 객체를 공유할 수 있어 자원 관리를 효율적으로 할 수 있다.

단점

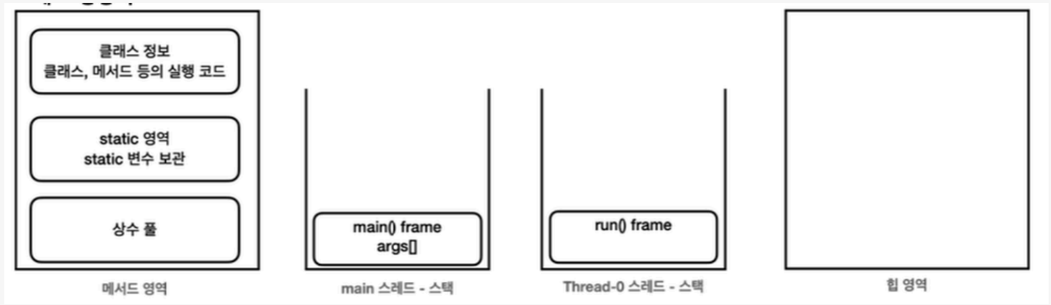
- 코드가 약간 복잡해질 수 있다. Runnable 객체를 생성하고 이를 Thread 에 전달하는 과정이 추가된다.

스레드 생성 전, 후



스레드 생성 전

- 자바는 실행 시점에 main 이라는 이름의 스레드를 만들고 프로그램의 시작점인 main() 메서드를 실행한다.
- main 스레드는 자바가 만들어주는 기본 스레드.

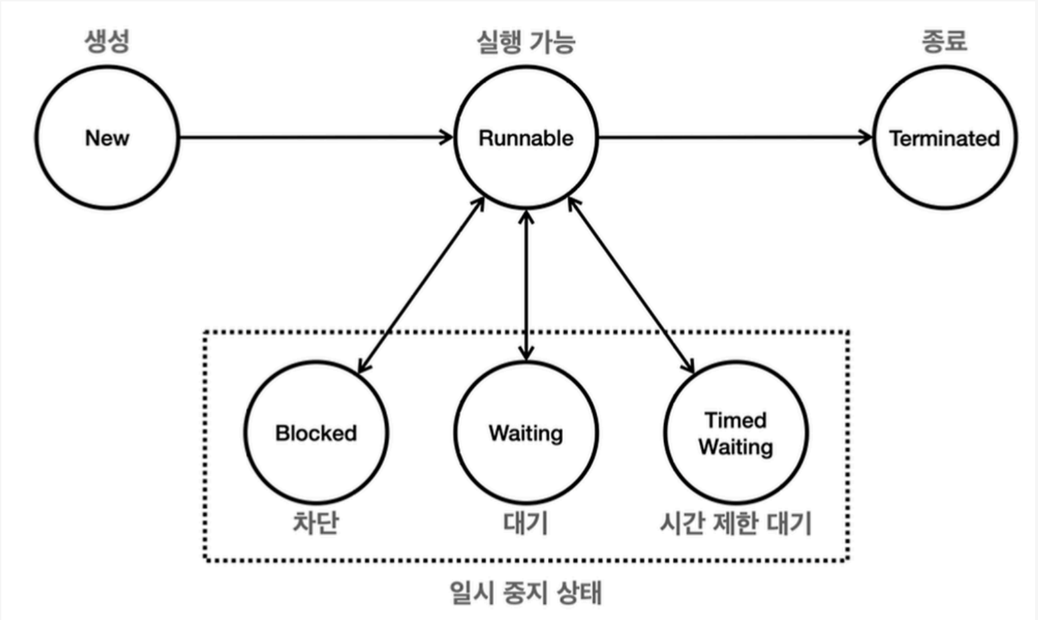


스레드 생성 후

- start() 메서드를 호출하면 자바는 스레드를 위한 별도의 스택 공간을 할당한다.
- start() 메서드를 호출하면 생성된 스레드가 run() 메서드를 실행한다.
- main 스레드가 run() 메서드를 실행하는게 아니라 Thread-0 스레드가 run() 메서드를 실행
- main 스레드는 start() 메서드 호출 후 바로 다음 줄 코드를 실행한다.

스레드 생명 주기

- New (새로운 상태): 스레드가 생성되었으나 아직 시작되지 않은 상태.
- Runnable (실행 가능 상태): 스레드가 실행 중이거나 실행될 준비가 된 상태.
- 일시 중지 상태들 (Suspended States)
  - Blocked (차단 상태): 스레드가 동기화 락을 기다리는 상태.
  - Waiting (대기 상태): 스레드가 무기한으로 다른 스레드의 작업을 기다리는 상태.
  - Timed Waiting (시간제한 대기 상태): 스레드가 일정 시간 동안 다른 스레드의 작업을 기다리는 상태
- Terminated (종료 상태): 스레드의 실행이 완료된 상태.



join

```java
public class JoinMainV3 {
    public static void main(String[] args) throws InterruptedException {
        SumTask task1 = new SumTask(1, 50);
        Thread thread1 = new Thread(task1, "thread1");

        thread1.start();

        // 스레드가 종료될 때 까지 대기
        log("join() - main 스레드가 thread1 종료까지 대기");
        thread1.join(1000);
        log("task1.result = " + task1.result);
    }

    static class SumTask implements Runnable {

        int startValue;
        int endValue;
        int result;

        public SumTask(int startValue, int endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }

        @Override
        public void run() {
            log("작업 시작");
            sleep(2000);
            int sum = 0;
            for (int i = startValue; i <= endValue; i++) {
                sum += i;
            }
            result = sum;
            log("작업 완료 result = " + result);
        }
    }
}
```

join 사용 X 결과

```
20:45:42.259 [     main] join() - main 스레드가 thread1 종료까지 대기
20:45:42.259 [  thread1] 작업 시작
20:45:42.265 [     main] task1.result = 0
20:45:44.268 [  thread1] 작업 완료 result = 1275
```

join 사용 O 결과

```
20:46:05.319 [     main] join() - main 스레드가 thread1 종료까지 대기
20:46:05.319 [  thread1] 작업 시작
20:46:07.334 [  thread1] 작업 완료 result = 1275
20:46:07.335 [     main] task1.result = 1275
```

join - 특정 시간만큼만 대기

```java
// 스레드가 종료될 때 까지 대기
log("join(1000) - main 스레드가 thread1 종료까지 1초 대기");
thread1.join(1000);
log("task1.result = " + task1.result);
```

```
21:01:20.546 [     main] join() - main 스레드가 thread1 종료까지 대기
21:01:20.546 [  thread1] 작업 시작
21:01:21.562 [     main] task1.result = 0
21:01:22.553 [  thread1] 작업 완료 result = 1275
```

인터럽트

```java
public class ThreadStopMainV2 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(4000);
        log("작업 중단 지시 thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

        @Override
        public void run() {
            try {
                while (true) {
                    log("작업 중");
                    Thread.sleep(3000);
                }
            } catch (InterruptedException e) {
                log("work 스레드 인터럽트 상태2 = " + Thread.currentThread().isInterrupted());
                log("interrupt message=" + e.getMessage());
                log("state=" + Thread.currentThread().getState());
            }
            log("자원 정리");
            log("자원 종료");
        }
    }
}
```

실행 흐름

- thread.interrupt() 호출 → 해당 스레드에 인터럽트 발생
- 스레드는 인터럽트 상태가 됨
- 인터럽트 상태에서 sleep() 호출 시 InterruptedException 발생
  - 스레드는 TIMED_WAITNG → RUNNABLE 상태로 변경
  - 인터럽트 상태에서 인터럽트 예외 발생 시 스레드의 인터럽트 상태는 종료됨

실행 결과

```
00:27:54.988 [     work] 작업 중
00:27:57.995 [     work] 작업 중
00:27:58.981 [     main] 작업 중단 지시 thread.interrupt()
00:27:58.990 [     main] work 스레드 인터럽트 상태1 = true
00:27:58.990 [     work] work 스레드 인터럽트 상태2 = false
00:27:58.991 [     work] interrupt message=sleep interrupted
00:27:58.992 [     work] state=RUNNABLE
00:27:58.992 [     work] 자원 정리
00:27:58.992 [     work] 자원 종료
```

인터럽트 - isInterrupted() 사용

```java
public class ThreadStopMainV3 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(100); // 시간을 줄임
        log("작업 중단 지시 thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

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

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

코드 변경

```java
while (!Thread.currentThread().isInterrupted()) { // 인터럽트 상태 변경X
    log("작업 중");
}
```

실행 흐름

- main 스레드에서 interrupt() 메서드를 호출해 work 스레드에 인터럽트를 건다.
- work 스레드는 인터럽트 상태라 isInterrupted()=true가 된다.
- while 조건이 false가 되면서 while문을 탈출한다.
- isInterrupted() 메서드는 인터럽트의 상태를 변경하지 않는다.
- 스레드의 인터럽트 상태가 true로 계속 유지되는 문제가 발생

실행 결과

```
                                    ......
01:12:20.490 [     work] 작업 중
01:12:20.490 [     work] 작업 중
01:12:20.490 [     work] 작업 중
01:12:20.490 [     main] 작업 중단 지시 thread.interrupt()
01:12:20.490 [     work] 작업 중
01:12:20.493 [     main] work 스레드 인터럽트 상태1 = true
01:12:20.493 [     work] work 스레드 인터럽트 상태2 = true
01:12:20.493 [     work] 자원 정리
01:12:20.494 [     work] 자원 정리 실패 - 자원 정리 중 인터럽트 발생
01:12:20.494 [     work] work 스레드 인터럽트 상태3 = false
01:12:20.494 [     work] 작업 종료
```

인터럽트 - interrupted() 사용

```java
public class ThreadStopMainV4 {

    public static void main(String[] args) {
        MyTask task = new MyTask();
        Thread thread = new Thread(task, "work");
        thread.start();

        sleep(100); // 시간을 줄임
        log("작업 중단 지시 thread.interrupt()");
        thread.interrupt();
        log("work 스레드 인터럽트 상태1 = " + thread.isInterrupted());
    }

    static class MyTask implements Runnable {

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

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

코드 변경

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

실행 흐름

- main 스레드에서 interrupt() 메서드를 호출해 work 스레드에 인터럽트를 건다.
- work 스레드는 인터럽트 상태라 Thread.interrupted()의 결과는 true가 된다
  - 이때 work 스레드의 인터럽트 상태를 정상(false)으로 변경한다.
- while 조건이 false가 되면서 while문을 탈출한다.

실행 결과

```
                                                .......
01:27:50.819 [     work] 작업 중
01:27:50.819 [     work] 작업 중
01:27:50.819 [     main] 작업 중단 지시 thread.interrupt()
01:27:50.819 [     work] 작업 중
01:27:50.822 [     main] work 스레드 인터럽트 상태1 = false
01:27:50.822 [     work] work 스레드 인터럽트 상태2 = false
01:27:50.822 [     work] 자원 정리
01:27:51.828 [     work] 자원 종료
01:27:51.829 [     work] 작업 종료
```

감사합니다.