🌒

Chapter 4. Thread & Concurrency (Part 1)

4.1 Overview

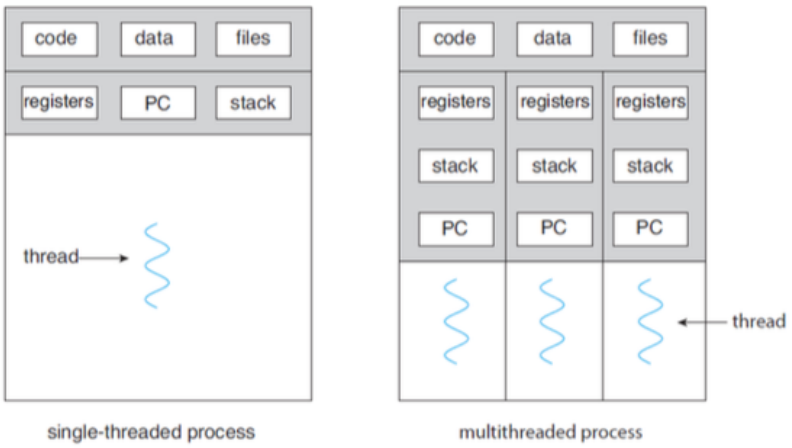
- 지금까지는 single thread 를 생각해봤다.
- 이제, 하나의 프로세스가 여려 개의 thread를 가질 수 있다는 것을 알게 되었다.

Thread

= lightweight process

= CPU를 점유하는 가장 기본적인 단위

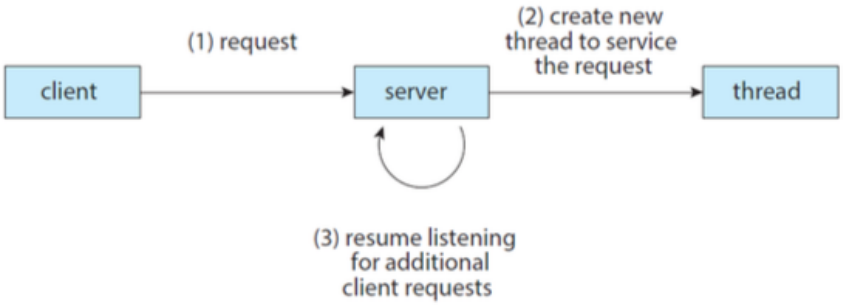
- thread ID가 CPU를 점유 한다 (pid 대신)
- thread id, pc, register set, stack 는 thread 별로 관리 해주어야 한다.



- single-threaded 가 여러 개 동작하는 것은 multiprocessing
  - thread가 여려개 동작하는 것은 multithreading
    - 멀티스레드 방식은 code, data, file은 서로 공유한다.
    - thread id, pc, register set, stack 는 별도로 관리 해주어야 한다.

multithreading의 좋은 점

- web server와 같은 client-server system



(1) client가 server에 요청한다.

(2) server는 client의 요청을 받아서 새로운 thread를 생성하여 요청을 thread가 하도록 시킨다.

(3) server는 nonblocking으로 다른 client의 요청을 받아들인다.

→ server는 만들 수 있는 한도 내에서 thread를 생성하여 client의 요청을 처리할 수 있다.

→ 대부분의 현대적인 소프트웨어에서 지원함

multithreading의 장점

1. **Responsiveness** : nonblocking으로 계속 실행할 수 있다.
   ㄴ process는 user interface같은 것을 처리할 때 block 상태에서 처리한다.
2. **Resource Sharing** : 스레드는 프로세스의 자원을 공유한다.
   ㄴ shared-memory(별도의 공간을 공유), message-passing(os에 있는 큐 사용) 방식보다 data공유가 쉽다.
3. **Economy** : 프로세스를 생성하는 것 보다, 스레드를 생성하는 것이 더 경제적이다.
   ㄴ 스레드 switching이 context switching(프로세스)보다 오버헤드가 적다.
   ㄴ context switching은 PCB등을 교환한다.
4. **Scalability** : 확장성, 멀티프로세스 아키텍쳐에서 강점이 된다. → 스레드를 여러개 붙여 병렬처리 가능

4.4 Thread Library

Java의 스레드

- 스레드는 자바로 쉽게 공부할 수 있다! → 스레드 기반으로 개발됨
- 자바 프로그램에서, 스레드는 프로그램 실행의 기본적인 모델이다.

- 자바에서 스레드의 생성과 관리를 위한 많은 도구를 제공한다.

Java에서 스레드를 만드는 3가지 방법

- Thread 클래스를 상속(Inheritance)받는 방법
  1. Thread 클래스부터 파생된 새로운 클래스를 생성한다.
  2. 그 클래스의 public void run() 메소드를 오버라이드 한다.

  ⇒ 이렇게 구현했을 때, 다중 상속이 안되는 것이 단점 > 아래 방법을 사용
- Runnable 인터페이스를 Implementing한다. → 가장 많이 쓰는 방법
  1. Runable 인터페이스가 구현된 새로운 클래스를 정의한다.
  2. 그 클래스의 public void run() 메소드를 오버라이드 한다.
- Lambda 표현을 사용한다. (익명 스레드) → 가장 간단한 방법
  1. 새로운 클래스를 정의하는 것 보다 낫다.
  2. Runnable 을 사용하는 대신 lambda expression을 사용한다.

방법1 : Thread 클래스 상속

```
class MyThread1 extends Thread {
  public void run(){
    try{
      while (true){
            System.out.println("Hello, Thread!");
            Thread.sleep(500); //0.5초
      }
    }
    catch (InterruptedException ie) {
      System.out.println("I'm interrupted");
    }
  }
}



public class ThreadExample1 {
  public static final void main(String[] args) {
    MyThread1 thread = new MyThread1(); //인스턴스 생성
    thread.start(); // start를 호출하면 run()메소드를 호출해 줌
    System.out.println("Hello, My Child!");
  }
}
```

실행결과

- `main` 이라는 스레드 하나가 실행하다가 > `start()` 메소드를 만나면, `MyThread1`의 `run()` 메소드를 실행시키며 > 새로운 스레드가 실행을 하게 된다.

- `start()`를 수행한 후, 아직 context switch가 일어나지 않았으므로, *"Hello, My Child!"*를 먼저 출력한다.

```
Hello, My Child!
Hello, Thread!
Hello, Thread!
Hello, Thread!
Hello, Thread!
...
```

방법2 : Runnable 인터페이스 구현하기

```
class MyThread2 implements Runnable {
  public void run(){
    try {
      while (true) {
          System.out.println("Hello, Runnable!");
          Thread.sleep(500);
      }
    }
    catch (InterruptedException ie) {
      System.out.println("I'm interrupted");
    }
}



public class ThreadExample2 {
  public static final void main(String[] args) {
    Thread thread = new Thread(new MyThread2() ); //Thread 클래스 생성자에 Runnable 스레드를 인자로 넘겨줌
    thread.start();
    System.out.println("Hello, My Runnable Child!");
  }
}
```

실행결과

→ 위와 같다.

방법3 : Runnable Rambda 표현식 사용하기

```
public class ThreadExample3{
  public static final void main(String[] args) {
      Runnable task = () -> {
          try {
            while (true) {
                  System.out.println("Hello, Lambda Runnable!");
            }
          }
          catch (InterruptedException ie) {
              System.out.println("I'm interrupted");
```

```
            }
        };

        Thread thread = new Thread(task);
        thread.start();
        System.out.println("Hello, My Lambda Child!");
    }
}
```

실행결과

- 새로운 클래스를 생성하는 대신, 함수형식으로 구현할 수 있도록 해준다.

→ 위와 같다.

예제1 :부모 쓰레드의 대기 : join

```
public class ThreadExample4{
    public static final void main(String[] args) {
        Runable task = () -> {
            for (int i = 0; i < 5 ; i++) {
                System.out.println("Hello, Lambda Runnable!");
            }
        };

        Thread thread = new Thread(task);
        thread.start();
        try {
            thread.join();
        }
        catch (InterruptedException ie) {
            System.out.println("Parent thread is interrpted");
        }
        System.out.println("Hello, My Joined Child!");
    }
}
```

- join의 역할 ?

실행결과

- `start()`를 만나면, `task`의 `run()` 메소드를 호출한다.
- `join()`을 만나면, `main` 스레드는 wait한다. > `task` thread 가 수행된다.
- `task`가 수행이 끝나면, `main` 스레드가 이어서 작업을 수행한다.

```
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, My Joined Child!
```

예제2 : 쓰레드의 종료 : interrupt

```
public class ThreadExample5 {
    public static final void main(String[] args) throws InterruptedException {
        Runnable task = () -> {
            try {
                while (true) {
                    System.out.println("Hello, Lambda Runnable!");
                    Thread.sleep(100);// 0.1초
                }
            }
            catch (InterruptedException ie) {
                System.out.println("I'm interrupted");
            }
        };
        Thread thread = new Thread(task);
        thread.start();
        Thread.sleep(500); //0.5초
        thread.interrupt();
        System.out.println("Hello, My Interrupted Child!");
    }
}
```

실행결과

- `task`(thread)는 `sleep(100)` 0.1초 마다 "Hello, Lambda Runnable!"을 출력한다.
- `main` 스레드는 `sleep(500)` 0.5초 후에 interrupt()를 호출 > "I'm interrupted" 을 출력한다.
- `main` 스레드는 나머지 부분을 이어서 수행한다.

```
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
Hello, Lambda Runnable!
I'm interrupted
Hello, My Interrupted Child!
```

4.2 Multicore Programming

Multithreading in a Multicore system

- **동시성(Concurrency)**을 향상시키기 위해 여러 개의 코어를 더 효율적으로 사용해야한다.
- 네 개의 스레드가 있는 애플리케이션이 있다고 가정

  ㄴ single-core : 스레드가 시간에 따라 배치된다. (interleaved : 사이사이에 끼워넣기 > 시분할 time sharing)

  ㄴ multiple-cores: 여러 스레드가 병렬(parallel)로 실행될 수 있다.

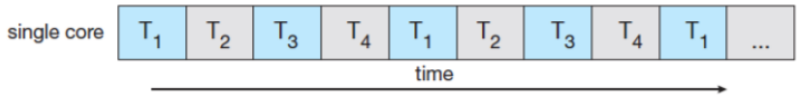


Figure 4.3 *Concurrent execution on a single-core system.*

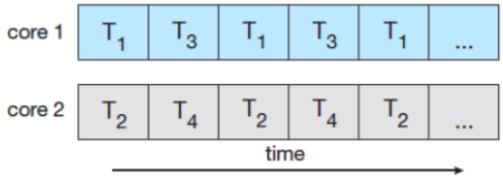


Figure 4.4 *Parallel execution on a multicore system.*

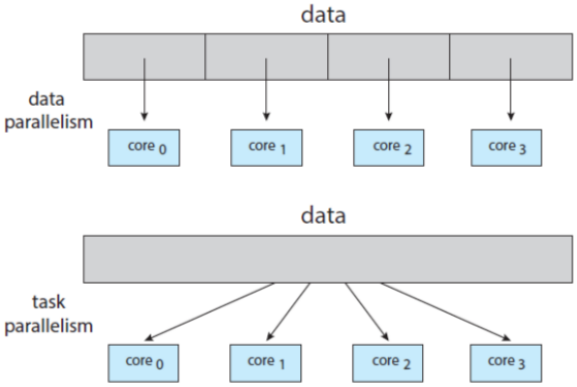
multicore system에서 프로그래머가 해야하는 일

- Identifying task : 어떤 task가 병렬로 실행이 가능할 지 구분해야 한다.

  ex) merge sort는 어떤 부분이 정렬되어야만 다음 부분과 합칠 수 있다.
- Balance : 각 코어가 같은 양의 작업을 수행할 수 있도록 벨런싱을 해야한다.
- Data splitting : 각 코어에서 수행할 수 있도록 data가 적절하게 나뉘어야 한다.
- Data dependency : 데이터가 작업 수행에 맞추어 동기화되어야 한다.
- Testing and debugging : 테스트와 디버깅이 single-thread에 비해 어렵다.

Types of parallelism

- 두 가지 타입의 병렬구조
  - 데이터를 나누거나
  - 작업을 나누거나
- 위 두개가 중요하지 않아졌다.
  - 클라우드를 사용한 분산시스템 등으로 인해



Amdahl의 법칙

- 암달의 법칙
- 코어는 많을 수록 좋은가?
- 모든 작업이 병렬처리 가능하다면 코어는 많을 수록 좋다. (그래프의 빨간선)
- 병렬처리 가능한 작업의 비율이 낮다면 코어가 많아도 소용이 없다.