스레드와 동시성 프로그래밍

동시성 (Concurrency)과 병렬성 (Parallelism)

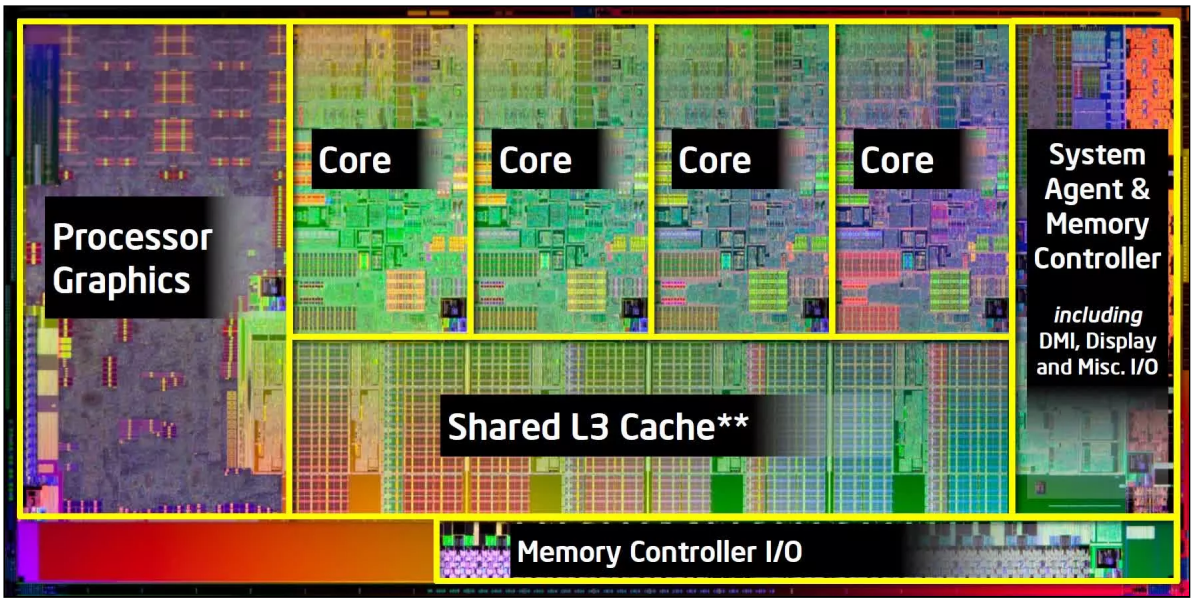
동시성

- 하나의 CPU를 이용해서도 **시분할**(time slicing)을 통해 **논리적인 동시성**을 달성 가능함
- 충분히 빠른 속도로 여러 프로그램을 바꾸어가며 실행(스케쥴링 + 컨텍스트 스위칭)하여 겉으로 보기에는 동시에 실행되는 것처럼 보이게 할 수 있음

병렬성

- 동시성이 **하드웨어 지원**을 통해 이루어짐 (단, 여러개의 **코어**를 가지고 있는 CPU가 필요함)
- **진정한 의미에서 동시에 실행**됨
    - 병렬성을 확보한 상태(멀티 코어 CPU 사용)에서 각각의 코어에서 동시에 여러 프로그램을 실행(동시성) 가능
- 동시에 여러 프로그램을 많이 사용하는 환경에서는 멀티 코어 CPU 사용할 경우 CPU 각각의 부하(workload)가 낮아짐
- 발열 문제로 CPU 클럭 속도(연산 속도)를 유의미하게 상승시키는 것이 기술적으로 어려워져서, 최근에는 사용할 수 있는 코어를 추가하는 쪽으로 발전 중

멀티 코어 CPU의 구조



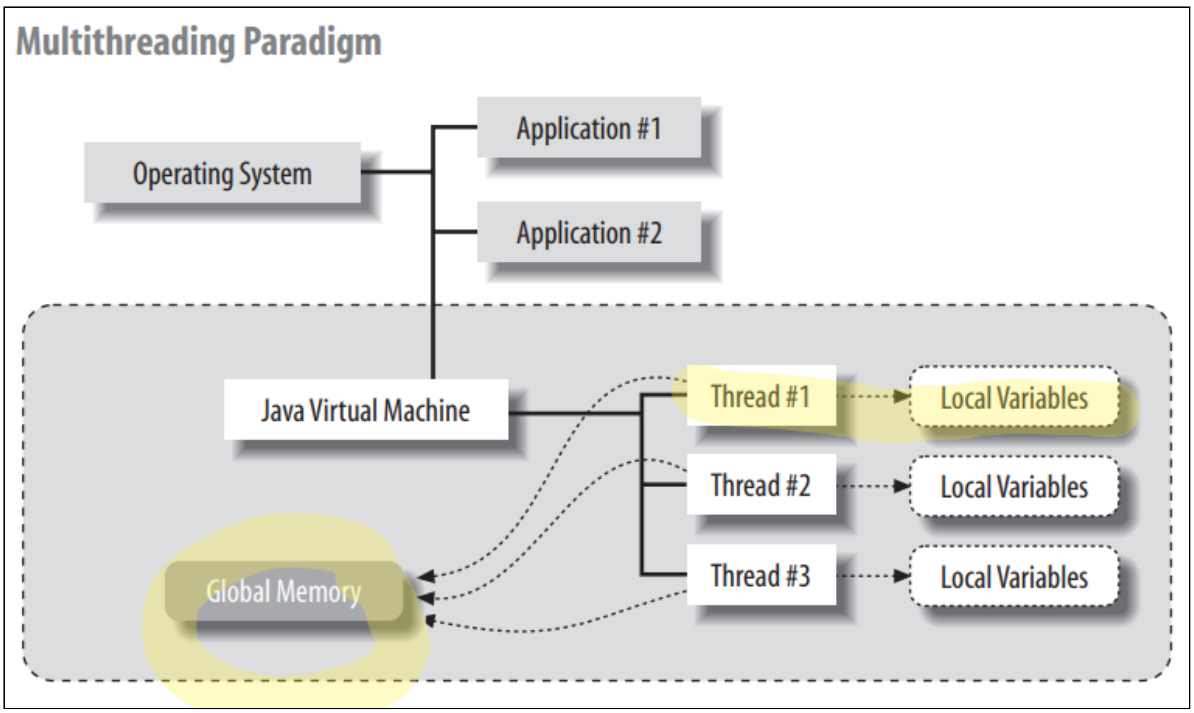
프로세스 vs 스레드

프로세스

- 프로그램 실행의 단위
- 프로그램 (**명령어들의 집합**, **실행하기 전**까지는 아무런 영향이 없음) => **실행 (exe 파일을 더블 클릭)** => **운영체제에서 프로세스를 생성하여 메모리를 할당하고 실제 명령어 집합을 순차적으로 실행**함
- 윈도우의 작업관리자의 프로세스 탭에서 현재 수행중인 프로세스들의 목록을 확인 가능
    - 프로세스는 일반적으로 병렬적(여러 코어를 사용)으로 실행됨 (최근 출시되는 CPU는 거의 대부분 코어 2개는 가지고 있음)

스레드

- **경량화된 프로세스(lightweight process)**
    - 프로세스 생성 및 관리 비용보다 스레드 생성 및 관리 비용이 더 적음
    - 프로세스 내부에서 사용할 **일부분의 명령어 집합을 동시에 실행**하도록 만들 수 있음
    - 프로세스가 이루어야 할 **공통의 목표를 위해서 분업 작업(task)**을 진행하도록 만들 수 있음
        - ex) 채팅 프로그램 (채팅 내용 입력 작업, 새로 전달된 메시지를 받아 출력하는 작업)
        - ex) 게임 프로그램 (키보드, 마우스 입력 작업, 화면 그리는 작업, 적의 행동 패턴 계산 작업, 그룹으로 부터 메시지 전달 여부 검사 작업, ...)
- 프로세스에서 생성된 스레드는 **프로세스의 메모리를 공유하며 및 통신**을 할 수도 있음
    - **프로세스 내에서 사용하는 데이터(변수, 객체)를 공유 가능**
        - 프로세스가 사용하는 **메모리(Heap)에 접근 가능**
    - 스레드 간 통신(wait, notify 메소드 사용) 가능
    - 보안 상의 이유로 프로세스간 메모리 공유는 기본적으로는 불가능 함
        - 단, **프로세스간 통신 기법(IPC, Inter-Process Communication)**을 이용하여 메모리 공유 및 통신을 할 수 있음



메인(main) 스레드

- 프로그램이 시작하면 자동으로 생성되는 스레드
    - 프로세스는 **최소 하나의 스레드**를 가지고 있음
- entry point 함수(**static void main(String[] argv)**) 메서드를 찾아 실행하는 스레드
- 다른 스레드가 필요하면 메인 스레드에서 여러 스레드(멀티스레드(Multi-thread))를 생성하여 시작(spawn)해줄 수 있음

스레드 생성 방법

1. Thread 클래스를 상속받아 run 메소드 구현
2. **Runnable 인터페이스를 구현하여 run 메소드 구현 (권장되는 방법)**
3. 익명 스레드, 익명 Runnable을 사용

스레드의 시작

- **run 메서드**를 직접 호출은 의미가 없음
    - 새로운 **실행 환경**(Call Stack)을 생성하지 않음
        - run 메소드로 실행시킬 경우 해당 메소드를 호출한 스레드의 Call Stack을 사용함
    - 그냥 일반적인 메소드 호출처럼 처리(순차적 실행)되며 동시성이 없음
    - 결론 => 스레드 자체를 생성하지 않음
- **start 메서드**를 통해 스레드 시작
    - start 메서드 호출이후 운영체제의 스레드 스케쥴러의 지시에 따라 실행 상태(runnable 상태)로 변하게 되고 내부 로직을 실행함
    - 각 스레드는 스레드만의 실행 환경(Call Stack)을 가짐

ThreadStartDemo.java

```
// 1.
// 스레드를 상속받는 스레드 클래스 생성하고
class MyThread extends Thread {
    // run 메서드 오버라이드해서 동시성 로직 작성
    @Override
    public void run() {
        System.out.println("run 1");
    }
}

// 2.
// Runnable 인터페이스를 구현하고
class MyRunnable implements Runnable {
    // run 메서드 오버라이드해서 동시성 로직 작성
    @Override
    public void run() {
        System.out.println("run 2");
    }
}
```

```
public class ThreadStartDemo {
    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        // myThread.run(); // run 메서드 호출시 스레드 생성하지 않고 그냥 메서드 실행하듯이 run 메서드가 실행됨
        myThread.start();

        // 스레드 생성자에 Runnable 객체 전달하여
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        // run 메소드 로직 실행
        thread.start();

        // 3.
        // 익명 객체 만들며 바로 시작
        (new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("run 3");
            }
        })).start();
    }
}
```

ThreadConcurrencyDemo.java

```
class CountRunnable implements Runnable {
    private int count;

    public CountRunnable(int count) {
        this.count = count;
    }

    @Override
    public void run() {
        for(int i=count;i>0;i--) {
            System.out.println(Thread.currentThread().getName() + " " + i);
            try {
                // 스레드 1초 정지시키기 (waiting 상태로 변경되며 다른 스레드에게 CPU 점유권을 양도)
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
        System.out.println("terminate thread");
    }
}

public class ThreadConcurrencyDemo {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new CountRunnable(10));
        Thread thread2 = new Thread(new CountRunnable(5));
        // 스레드 이름 변경
        thread1.setName("Thread 1");
        thread2.setName("Thread 2");
        thread1.start();
```

```
        thread2.start();
    }
}
```

스레드 우선순위

- 스레드 스케줄러는 각 스레드에 실행 시간을 할당해주는데 얼마나 많은 시간을 할당해주느냐는 **우선순위**에 따라 달라짐
    - 하지만 절대적이라고는 할 수 없으며 JVM 구현에따라, 운영체제 스케줄러 정책에 따라서 시간 할당 정도가 달라질 수 있음
- 1 ~ 10까지 지정 가능하며 미리 지정해 놓은 상수도 있음
    - MIN_PRIORITY : 1
    - NORM_PRIORITY : 5
    - MAX_PRIORITY : 10
- 기본적으로 스레드 생성시 우선순위는 NORM_PRIORITY

ThreadPriorityDemo.java

```java
public class ThreadPriorityDemo {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=0;i<1000;i++) System.out.print("1");
            }
        });
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                for(int i=0;i<1000;i++) System.out.print("2");
            }
        });

        thread1.setPriority(10);
        thread2.setPriority(1);

        thread1.start();
        thread2.start();
    }
}
```

스레드 종료

- run 메서드가 종료되면 스레드도 종료됨
- run 메서드 실행 도중에 스레드를 종료하고 싶을 경우 일반적인 스레드 종료 처리 방법은 다음의 두 방식
    1. **종료 플래그 변수**의 상태를 확인하여 종료 (단, 이 경우 **volatile 키워드** 사용 필요)
    2. **인터럽트 신호**를 보내서 종료

- deprecated된 stop 메소드는 사용 금지

TerminationFlagDemo.java

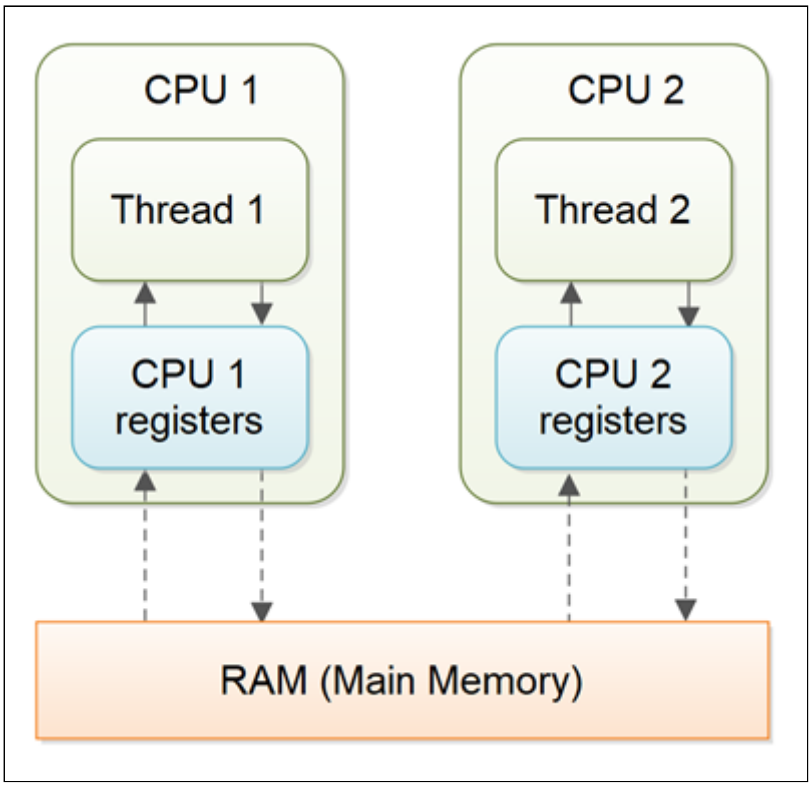
```
class TerminatableRunnable implements Runnable {
    private volatile boolean terminate = false;

    @Override
    public void run() {
        while(!terminate) {
            System.out.println("running...");
        }
        System.out.println("terminated");
    }

    public void terminate() {
        this.terminate = true;
    }
}

public class TerminationFlagDemo {
    public static void main(String[] args) {
        TerminatableRunnable terminatableRunnable = new TerminatableRunnable();
        (new Thread(terminatableRunnable)).start();

        (new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
                terminatableRunnable.terminate();
            }
        })).start();
    }
}
```

ThreadInterruptDemo.java

```
class InterruptableRunnable implements Runnable {

    @Override
    public void run() {
        while(true) {
            boolean isInterrupted = Thread.currentThread().isInterrupted();
            if(!isInterrupted) {
                System.out.println("running... " + isInterrupted);

                // waiting 상태에 진입했을 때 interrupt가 생기면
InterruptedException 예외 발생
                /*
                try {
                    Thread.sleep(100);
```

```
                } catch (InterruptedException e) {
                    System.out.println("InterruptedException!");
                    break;
                }
                */
            } else {
                break;
            }
        }
    }
}

public class ThreadInterruptDemo {
    public static void main(String[] args) {
        Thread thread = new Thread(new InterruptableRunnable());
        thread.start();

        (new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
                System.out.println("call interrupt from another thread!");
                thread.interrupt();
            }
        })).start();
    }
}
```

volatile 키워드



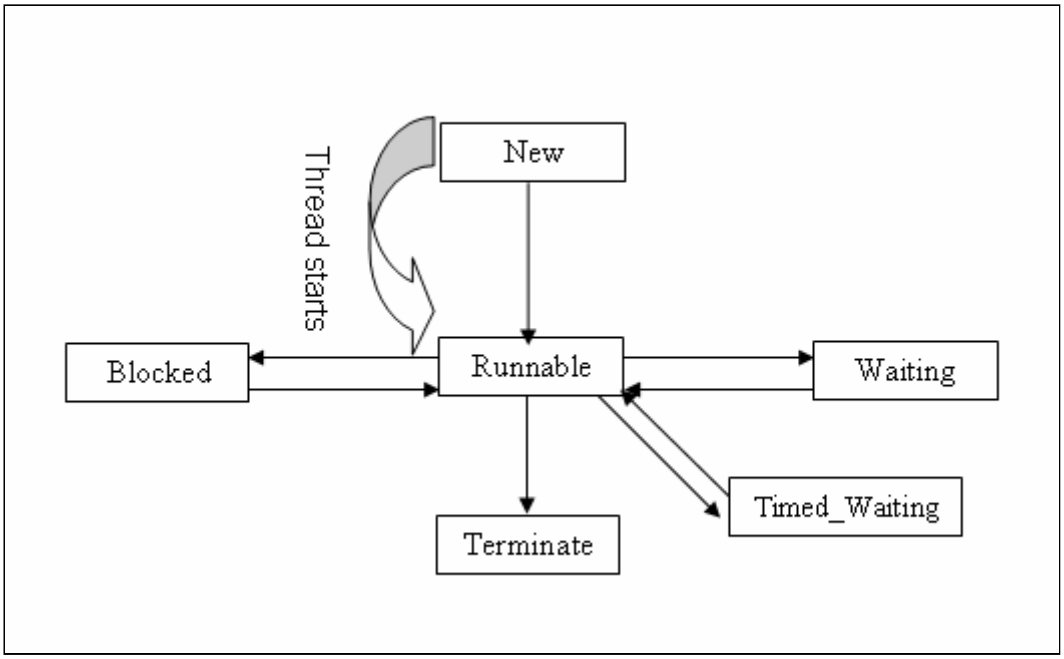
- 스레드 간 공유해야 할 변수 값이 **캐시에 의해 스레드 간 보이지 않는 상황을 막기 위해서 특정 변수가 메인 메모리에서 동기화되도록 강제 유도**하는 키워드 (visibility guarantee)
    - 캐시를 통한 변수 접근이 메인 메모리를 통한 변수 접근보다 더 빠르기 때문에, 변수 접근시 **퍼포먼스 하락**을 감수해야 함

VolatileDemo.java

```
class StoppableRunnable implements Runnable {
    // volatile 키워드가 붙었을 때, 없을 때 차이 살펴보기
    // private volatile boolean stop = false;
    private boolean stop = false;

    @Override
    public void run() {
        long count = 0;
        while(!stop) {
            count++;
        }
        System.out.println(count);
        System.out.println("exit run");
    }

    public void stop() {
        // stop 값 변경 후 확인을 위해서 출력
        this.stop = true;
        System.out.println("stop runnable " + this.stop);
    }
}

public class VolatileDemo2 {
    public static void main(String[] args) {
        StoppableRunnable runnable = new StoppableRunnable();
        new Thread(runnable).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
                // 1초 이후 stop 메서드를 호출하여 내부 stop 값을 true로 변경
                runnable.stop();
            }
        }).start();
    }
}
```

- https://javarevisited.blogspot.com/2017/01/can-we-make-array-volatile-in-java.html

스레드의 상태



상태	설명
new	스레드를 객체를 생성만하고 start를 메소드를 호출하지 않아 아직 시작되지 않은 상태
runnable	run 메서드의 코드를 실행중인 상태
blocked	스레드가 동기화된 **synchronized 키워드로 동기화 된 코드 블록 혹은 메소드**에 접근하였으나, 이미 해당 메소드를 실행하고 있는 스레드가 있어 대기하고 있는 상태
waiting	**(무기한)** 다른 스레드가 깨워주기(wait => notify)를 기다리거나 특정 스레드가 종료(join)되기를 기다리면서 대기하는 스레드 상태
timed_waiting	**(특정 시간동안)** sleep 메서드를 호출하거나, 다른 스레드가 깨워주기(wait => notify)를 기다리거나 특정 스레드가 종료(join)되기를 기다리면서 대기하는 스레드 상태
terminated	**run 메서드가 종료**되어 모든 작업이 마무리 된 스레드 상태

스레드 정보 및 상태 관련 API

```
- Thread Thread.currentThread : 현재 실행되고 있는 스레드 리턴 (static 메서드임을 주목)
- void setName(String) : 스레드의 이름 정하기
- String getName : 스레드의 이름 리턴
- int getPriority : 스레드의 우선순위 리턴
- long getId : 고유한 스레드 ID(숫자값) 리턴 (단, 종료된 스레드의 ID는 추후 시작되는 스레드에서 사용될 수 있음)
- State getState : 스레드의 상태 리턴
```

- 멀티 코어에서 병렬적으로 진행되는 스레드가 있다면 currentThread는 어떻게 동작하는가?

스레드 활용 사례

채팅 프로그램 (클라이언트)

- 사용자 입력 스레드 : 사용자의 입력을 받아서 서버로 전달하는 역할을 수행
- 메시지 출력 스레드 : 새로 전송된 메시지를 확인 후 화면에 출력하는 역할을 수행

채팅 프로그램 (서버)

- 리스너 소켓 스레드 : 새로운 연결을 위해서 대기하는 스레드 (안내데스크)
- 클라이언트 소켓 스레드 : 새로운 연결을 처리하기 위해서 새로 생성되는 스레드 (종업원, 수리기사)

게임 프로그램

- 사용자 입력 스레드 : 사용자의 입력(키보드, 마우스)의 입력을 받고 적절히 처리
- 논리 스레드 : 게임의 로직(플레이어의 이동, 적의 이동, AI 등등)을 관리
- 그리기 스레드 : 화면에 이미지를 그려주는 역할을 수행

> 이 경우 논리 스레드와 그리기 스레드를 합쳐도 무방함(동시에 진행되지 않고 순차적으로 진행되어도 문제 없으므로), 그러나 사용자 입력 스레드는 빠른 반응성을 확보하기 위하여 반드시 분리 필요!

안드로이드

- 네트워크 통신(ex: 파일 다운로드, 웹 페이지 정보 요청, 웹 API 호출 등등)을 위한 스레드 생성(AsyncTask) : 만약 스레드를 이용하지 않고 호출할 경우 ANR(Application Not Responding) 유발 가능
- 메인 스레드에서는 GUI에 대한 요청 작업을 처리하고 오래 걸리는 작업은 전부 스레드로 분리시켜 처리하고 결과를 통보받는 방식으로 구현

병렬 처리의 방식

- 태스크 분해 : **작업 할당과 분업**에 초점을 맞춘 방식
  - 예) 한 스레드에서는 뉴스 데이터를 가져오는 역할을 수행, 한 스레드에서는 가져온 뉴스 데이터의 키워드를 분석하는 역할을 수행
- 데이터 분해 : **병렬성을 활용한 효율성**에 초점을 맞춘 방식
  - 예) 100만개의 요소를 가진 배열의 합 구하기, 이미지의 픽셀들 화면에 출력하기, 영상 인코딩, 압축 등등